University of California
Santa Barbara

Software Side-Channel Analysis

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Lucas A. Bang

Committee in charge:

Professor Tevfik Bultan, Chair
Professor Ömer Eğecioğlu
Professor Ben Hardekopf

June 2018

The Dissertation of Lucas A. Bang is approved.

Professor Ömer Eğecioğlu

Professor Ben Hardekopf

Professor Tevfik Bultan, Committee Chair

June 2018

Software Side-Channel Analysis

by

Lucas A. Bang

Acknowledgements

First and foremost, I thank my Ph.D. research advisor, Tevfik Bultan. I am grateful for Tevfik's guidance, encouragement, and feedback, which were crucial over the many years I spent in the Verification Lab at UCSB. He is an outstanding advisor, mentor, and teacher who is always happy to share his academic expertise and experience to ensure the success of his students.

Many thanks to Ömer Eğecioğlu and Ben Hardekopf for being on my committee, taking the time to attend my graduate program milestone presentations, and providing feedback and encouragement throughout the steps of finishing my Ph.D.

Lisa Berry is a fantastic instructor who taught me so much about how to be an effective teacher and always strive to learn new tools to add to my teaching toolbox. Tim Sherwood's mentorship on teaching, research, and academia was especially helpful during my first years at UCSB.

My research at UCSB could not have happened without collaborations with many hard-working, brilliant, and inspiring researchers. I am especially thankful to have collaborated closely with Baki Aydin and Nicolas Rosner. Collaborating with Corina Psreanu and Sang Phan during my internship at CMU and afterward during our work in the STAC program was an inspiring and formative experience.

I will always look back fondly on the time I spent with the people in the Verification Lab. Bo, Baki, Tegan, Miroslav, Will, Seemanta, Burak, Nestan, Nicolas, and Isaac have all been such wonderful company. Our wide-ranging conversations, whether serious, hilarious, or downright absurd, were always something for me to look forward too when heading into the lab.

Finally, I am ever grateful for my amazing partner, Athena, whose steadfast, unwavering, loving support has been invaluable.

Curriculum Vitæ

Lucas A. Bang

Education

2018	Ph.D. in Computer Science, University of California, Santa Barbara.
2013	M.S. in Computer Science, University of Nevada, Las Vegas.
2010	B.A. in Computer Science, University of Nevada, Las Vegas.
2010	B.S. in Mathematics, University of Nevada, Las Vegas.

Publications

Lucas Bang, Nicolas Rosner, and Tevfik Bultan. "Online Synthesis of Adaptive Side-Channel Attacks Based On Noisy Observations." Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P 2018).

Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. "Synthesis of Adaptive Side-Channel Attacks." Proceedings of the 2017 IEEE Computer Security Foundations Symposium (CSF 2017).

Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. "String Analysis for Side Channels with Segmented Oracles." Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2016).

Lucas Bang, Abdulbaki Aydin, and Tevfik Bultan. "Automatically Computing Path Complexity of Programs." Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015).

Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. "Automata-Based Model Counting for String Constraints." Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015).

Lucas Bang, Wolfgang W. Bein, Lawrence L. Larmore. "R-LINE: A Better Randomized 2-server Algorithm on the Line." Theoretical Computer Science (TCS). Volume 605, 2015.

Lucas Bang, Wolfgang W. Bein, Lawrence L. Larmore. "R-LINE: A Better Randomized 2-Server Algorithm on the Line." Proceedings of the 10th Workshop on Approximation and Online Algorithms (WAOA 2012).

Abstract

Software Side-Channel Analysis

by

Lucas A. Bang

Software side-channel attacks are able to recover confidential information by observing non-functional computation characteristics of program execution such as elapsed time, amount of allocated memory, or network packet size. The ability to automatically determine the amount of information that a malicious user can gain through side-channel observations allows one to quantitatively assess the security of an application. Since most software that accesses confidential information leaks some amount of information through side channels, it is important to quantify the amount of leakage in order to detect vulnerabilities. In addition, one can prove that a program is vulnerable to side-channel attacks by synthesizing attacks that recover confidential information.

In this dissertation, I provide methods for (1) quantifying side-channel vulnerabilities and (2) synthesizing adaptive side-channel attack steps. My approaches advance the state-of-the-art in automatic software side-channel analysis which I summarize as follows. I make use of symbolic execution to extract program constraints that characterize the relationship between secret information, the inputs of a malicious user, and observable program behaviors. By applying model counting constraint solving to these constraints, I compute probabilistic relationships among secrets, attacker inputs, and attacker side-channel observations. These probabilities are used to quantify information leakage for a program by applying methods from the field of quantitative information flow. Moreover, by automatically generating a symbolic expression that quantifies information leakage, I am able to perform numeric maximization over attacker inputs to synthesize optimal

attack steps. The sequence of attack steps serves as a proof of exploitability. I give two different automatic attack synthesis techniques: a fully static approach and an online dynamic approach that constructs an attack that takes into account system noise and is able to execute the attack through the network. I demonstrate the effectiveness of my approaches on a set of experimental benchmarks.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Since computers are used in every aspect of modern life, many software systems have access to secret information such as financial and medical records of individuals, trade secrets of companies, and military secrets of states. Confidentiality, a core computer security attribute, dictates that a program that manipulates secret information should not reveal that information. This can be hard to achieve if an attacker is able to observe different aspects of program behavior such as execution time and memory usage. Side-channel attacks recover secret information from programs by observing non-functional characteristics of program executions such as time consumed, memory accesses, or packets transmitted over a network [3, 4, 5, 6, 7]. The Spectre and Meltdown attacks, two of the biggest security vulnerabilities in recent history, are side-channel attacks which make observations on cache usage and use inferences based on speculative execution to steal secret information [8, 9].

I will begin with an informal answer to the question "What is a side channel?" In August 1990, Time Magazine published an article, "And Bomb The Anchovies" regarding an information leak about the work happening inside the United States Pentagon [10, 11].

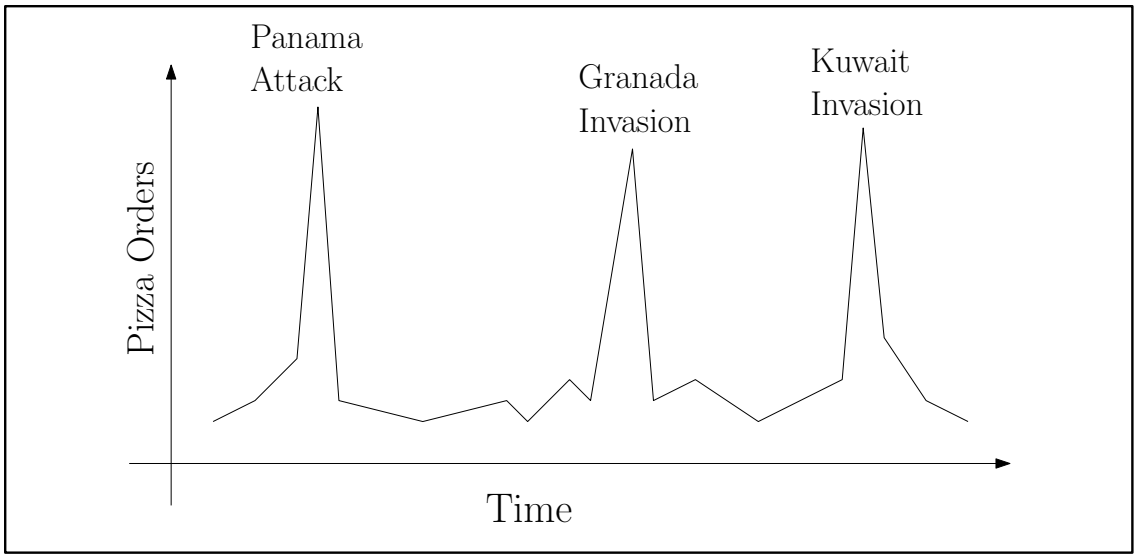


Figure 1.1: The number of pizzas ordered by the Pentagon spikes up around the time of major political events (note: this figure not based on actual data).

> Delivery people at various Domino's pizza outlets in and around Washington claim that they have learned to anticipate big news baking at the White House or the Pentagon by the upsurge in takeout orders. Phones usually start ringing some 72 hours before an official announcement. "We know," says one pizza runner. "Absolutely. Pentagon orders doubled up the night before the Panama attack; same thing happened before the Grenada invasion." Last Wednesday, he adds, "we got a lot of orders, starting around midnight. We figured something was up." This time the big news arrived quickly: Iraq's surprise invasion of Kuwait.

Those at the Pentagon would like to believe that they are not leaking information about their future plans. Indeed, it is extremely difficult to learn exactly what they are up to, since it would require some form of *direct observation* of the secret information. On the other hand, by making *indirect observations* (the number of pizzas ordered), one is able to infer partial information—that a major geo-political event is about to

happen, not necessarily *what* the event will be. (I have heard that both the Pentagon and the Whitehouse have since opened internal 24-hour pizza kitchens so that their pizza consumption is no longer externally measurable.)

The Pentagon-Pizza side channel is an example of how the correlation between secret information and an unexpectedly observable signal can result in a vulnerability. Next, I will provide a brief tour through the history of side-channel vulnerabilities as they relate to electrical and computational processing of secret information. Arguably, the first documented electromagnetic signal-based side-channel vulnerability was the TEMPEST attack, the codename of a vulnerability kept secret by the National Security Agency until 2007 when it was partially declassified [12].

The TEMPEST vulnerability was discovered during World War II. Bell Labs had developed a teletype device for sending secure messages and sold it to the military. An operator could type the plain text of the message, it would be encrypted, and then sent to the recipient, who could then decrypt it using an agreed-upon system of cryptographic keys. Bell Labs had claimed that the device was guaranteed to be secure. However, a research engineer happened to notice that a nearby oscilloscope would display a faint signal each time the encryption device took a step. The operation of the encryption device included several electromechanical relays and switches. As was eventually discovered, the small bursts of electrical radiation generated by these components was measurable at least 100 feet away from another building. Even worse, the signals could be reliably correlated with the operators original teletype keystrokes!

Since that time, various other forms of side-channel vulnerabilities have been discovered. The acoustic emanations of dot-matrix printers—still used widely in doctor and legal practices for their high reliability—can be correlated with the content of the resulting printed page [13]. Inter-keypress timing differences can be measured by a network eavesdropper to determine an SSH user's encrypted password [14]. I was surprised to

read of a vulnerability in which a person's hands typing on a keyboard can distort the wifi signal from their laptop enough so that somebody with very inexpensive radio equipment can determine their keystrokes from a distance with high accuracy [15].

While these side channels are certainly *related* to the software that runs the printer, sends SSH packets, or controls the wifi card, they are more the result of the physical characteristics of the device or the predictable behavior of the human operator. Other side-channel vulnerabilities result from measurable hardware properties while performing specific computations. For instance, by measuring power usage, Paul Kocher showed that one can extract secret keys from a cryptographic device pair (e.g. a smart card and reader), since different instructions executed by the microprocesser have different power usage profiles. Measuring these profiles with standard signal processing equipment can reveal cryptographic keys used during DES, AES, and RSA encryption [16].

In this dissertation, I will specifically address how software implementations can be analyzed to discover and analyze side-channel vulnerabilities. For example, in Chapter 4 I will analyze implementations of password checking functions and the LZ77 compression algorithm. These two programs exemplify a type of side-channel vulnerability known as a *segment oracle*. At a high level, these side channels result from developers attempting to optimize code. In the case of the password checker, the function returns as soon as it knows that a password guess does not match. This results in a correlation between the execution time of the function and the size of the matching prefix of a user input, which can be exploited to efficiently reveal passwords [17, 18, 6]. This type of behavior is also present in the C implementation of `memcmp`, which was shown the be exploitable to leak hashed message authentication codes in the XBox—a serious security flaw [5]. The LZ77 compression algorithm, on the other hand, attempts to optimize the length of a message to be sent. This results in a correlation between the resulting message size and the length of a matching segment of attacker injected input [19]. This was also a serious

security flaw called Compression Ratio Info-leak Made Easy (CRIME) [20]. These are two intuitive examples of software side-channel vulnerabilities, and other programs are analyzed in later chapters.

The techniques I will describe in this dissertation use symbolic execution for the systematic analysis of program behaviors under different input values [21, 22, 23]. Furthermore, I use model counting [24, 25] over the constraints collected with symbolic execution to quantify the leakage of the detected side channels. Leakage quantification is accomplished by combining the results of model counting with concepts from information theory [26, 27]. The major contribution of this dissertation is to go beyond the current state-of-the-art in quantitative information flow (QIF). By casting the QIF problem as an objective function maximization problem over attacker interactions, one can synthesize side-channel attacks that cause the greatest amount of information to flow from secret program values to the attacker through side-channel observations. This allows me to demonstrate the vulnerability of a program, which brings me to my statement of contributions.

1.1 Contributions

By combining static program analysis, parameterized model counting, and optimization techniques, I quantify a program's vulnerability to side-channel attacks. Furthermore, I present a method that synthesizes an adaptive side-channel attack for a given function which demonstrates the functions's vulnerability. I make the following claims as novel research contributions contained in this dissertation.

1. **Automata-based model counting.** I developed automata-theoretic methods for counting the number of solutions to string and arithmetic constraints. I give a solution based on methods from algebraic graph theory, enumerative combinatorics,

and generating functions. I implemented that technique as part of a model counting constraint solving tool, ABC (Automata-Based model Counter). Model counting is a crucial component in quantitative program analysis.

2. **Quantitative information flow analysis for segment oracles.** One type of vulnerability is known as a segment oracle side channel. In Chapter 4, I describe a technique for automatically performing QIF for programs with this type of vulnerability. I give a combinatorial expression that can be used to efficiently quantify information leakage from segment oracle observations.

3. **Offline static side-channel attack synthesis.** One can demonstrate a program's vulnerability to side-channel attacks by providing an attack. I give a technique for synthesizing an attack tree that an adversary can use to exploit a side channel. Furthermore, I quantify the expected information gain for the attack. My approach makes use of symbolic representations of program and attacker behaviors, symbolic model counting, and numeric optimization.

4. **Online dynamic attack synthesis for noisy side channels.** The static attack-synthesis approach cannot account for the dynamic and noisy behavior of real systems. This final contribution is a method for synthesizing side-channel attacks that take into account system and network noise. I implemented a system that automatically generates a symbolic information-theoretic objective function that is numerically maximized in order to synthesize optimal attacker inputs. The ability to synthesize an attack for a function running on a live server proves the vulnerability of the function.

1.2 Dissertation Outline

In Chapter 2, I cover symbolic program analysis and the relevant ideas from quantitative information flow (QIF). I provide examples of symbolic QIF and demonstrate how it can be used to compare the security of different functions with respect to side-channel information leakage.

Chapter 3 covers techniques for model counting, enabling one to count the number of solutions for the constraints generated from symbolic execution. Counting constraint solutions allows one to compute probabilities of program behaviors. Those probabilities lead directly into methods for quantifying information flow of probabilistic systems. My earliest research at UCSB was in model counting for string constraints and this is the primary focus of the chapter. In addition, I discuss model counting for Boolean formulas (to server as a warm-up for more complex constraints), as well as model counting for integer constraints.

In Chapter 4, I present specialized techniques for segmented oracle side channels in which an attacker is able to explore each segment of a secret, for example each character of a password, independently. This technique can answer questions such as "what is the probability of discovering a password in k runs?" or "what is the leakage (in the number of bits) after k runs?" through side channels.

Quantifying information leakage is useful for measuring a program's vulnerability to side-channel attacks. However, I would like to answer questions like "what can an optimal attacker learn from the side channel?" In order to address these types of questions, in Chapter 5, I provide a method for synthesizing adaptive side-channel attacks, which reduces the attack synthesis problem to an information-theoretic optimization problem. This approach assumes an ideal discrete model of side-channel measurements. The resulting attack is a precomputed decision tree with branch decisions based on attacker-

controlled inputs and the corresponding idealized system observations. The attack tree tells the attacker what interaction to make with the system based on previous interactions and side-channel observations.

The attack synthesis framework described in Chapter 5 attempts to synthesize an attack strategy for all possible secret values. This quantification over all secret values results in scalability issues. However, in realistic scenarios, a system under attack can be considered to have a single, constant secret value (like a persistent database entry) that an attacker would like to reveal. Thus, an attacker can synthesize attack steps in an adaptive online fashion which eventually reveal that secret, rather than precomputing an entire attack tree over all secret values. Intuitively, this corresponds to an attacker performing an online discovery of a single path in the attack tree which is consistent with the unknown secret. However, in a real system, side-channel observations are far from ideal and are perturbed by noise (like random network delays). Thus, in Chapter 6, I provide a method for online adaptive attack synthesis in the presence of noisy observations. I make use of symbolic model counting, numeric optimization, and Bayesian belief updating to solve the side-channel attack synthesis problem. Finally, I discuss related work and I make observations on the application of these techniques in the final two chapters.

Chapter 2

Program Analysis for Quantitative Information Flow

In this chapter, I first give a high-level idea of what software side channels are and then discuss how symbolic execution is used to analyze a program. I then discuss probabilistic symbolic execution and how it can be used to quantify a program's vulnerability to side-channel attacks. I give some background on information theory and show how to apply it to the side-channel quantification problem.

2.1 Software Side Channels

In order to build up some intuition for software side-channel analysis, I will first give an informal discussion accompianied by an example. The reader may refer to Figure 2.1. We will consider systems in which a program P takes as input some *high-security* value h and a *low-security* value l. The value of h is intended to be secret and not publicly accessible. We shall assume that the low security input is under the control of a malicious attacker, $\mathcal{A}$. As an example, consider the code in Figure 2.2, and suppose that h

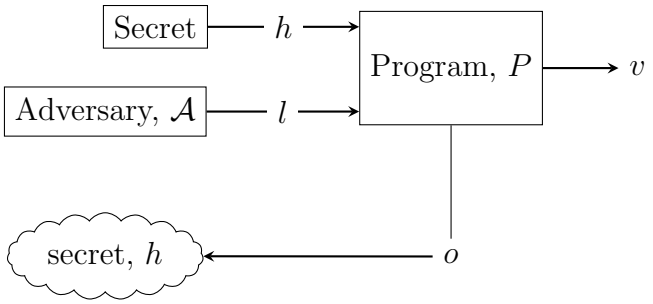


Figure 2.1: Model of the attacker $\mathcal{A}$, program P, secret input h, attacker-controlled input l, and side-channel observation o. The cloud represents the knowledge about the secret that the attacker gains by making input l and observing o.

represents an ATM PIN.

When an attacker $\mathcal{A}$ runs the system by providing input l (a guess for a PIN), the system accesses the secret input h (the stored PIN in the database). The program then performs some computation (comparing the input PIN to the stored PIN digit by digit), and outputs a value v (like true, the PINs match, or false, the PINs do not match). However, we will suppose that the attacker is able to observe some characteristics of the program execution, like running time. We call this the side-channel observation o.

Now, assume that the attacker knows the source code of the application, but does not know the secret. By performing analysis on the code, the attacker can determine correlations between the secret input h, the attacker's input l, and the side-channel observation o. Thus, by observing o the attacker can make inferences at h.

Returning to the example code in Figure 2.2, observe that there is a loop which compares h and l as arrays, element by element. Inside that loop, if a mismatch is ever detected, the function immediately returns `false`. Consequently, the running time of the function is correlated with the length of the longest matching prefix shared between h and l. When an attacker runs the program, and measures the running time, a longer running time indicates a longer prefix match with the unknown secret value. So, although the program appears safe, by only returning `true` for a complete match or `false` for any

```
1 public Boolean checkPIN(int[] h, int[] l){
2   for(i = 0; i < 4; i++){
3     if(h[i] != l[i])
4       return false;
5   }
6   return true;
7 }
```

Figure 2.2: A PIN-checking function.

```
1  public Boolean checkPIN(int[] h, int[] l){
2    Boolean matched = true;
3    for(i = 0; i < 4; i++){
4      if(h[i] != l[i]){
5        matched = false;
6      } else {
7        matched = matched;
8      }
9    }
10   return matched;
11 }
```

Figure 2.3: A PIN checking function that does not have a segment oracle side channel.

incomplete match, additional information about the prefix of h is leaked through running time measurements.

In fact, the above described side-channel vulnerability is us due to a common pattern. When comparing two pieces of data in the form of sequences, the individual elements are compared one at a time until the first mismatch is detected. This pattern exists in Java's `Arrays.equal()` and `String.equal()` functions, C's `memcmp` library function, as well as Python and JavaScript implementations of array and string equality and order comparisons. These implementations have lead to real vulnerabilities in the Xbox and the OAuth framework used by Google and Facebook [5, 28, 6, 18]. These patterns of side-channel vulnerabilities, in which side-channel measurements reveal information about prefixes of secrets, are known as *segmented oracle channels*. I give an extensive treatment of QIF for segmented oracle side channels in Chapter 4. Now that we have gained some intuition for side channel vulnerabilities, I begin our discussion of how to perform automatic side channel analysis of software.

2.2 Symbolic Execution

In order to perform automatic side-channel analysis, we need a way to automatically reason about the possible behaviors of programs. While there are many program analysis techniques at our disposal, I make use of symbolic execution throughout this dissertation. Symbolic execution [21] is a static analysis technique by which a program is executed on *symbolic* (as opposed to concrete) input values which represent all possible concrete values. If the set of actual variables of the program is $\{x_1, \ldots, x_n\}$, we associate a symbolic variable X_i to each variable x_i. Symbolically executing a program yields a set of *path constraints* $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$. A path constraint is a logical formual over symbolic program variables and we write $\phi_i(X_1, \ldots, X_n)$. Each ϕ_i is a conjunction of constraints on the symbolic inputs that characterize all concrete inputs that would cause a path to be followed. All the ϕ_i's are disjoint. Whenever symbolic execution encounters a branch condition c, both branches are explored and the constraint is updated: ϕ becomes $\phi \wedge c$ in the *true* branch and $\phi \wedge \neg c$ in the *false* branch. Path constraint satisfiability is checked using constraint solvers such as Z3 [29]. If a path constraint is found to be unsatisfiable, that path is no longer analyzed. For a satisfiable path constraint, the solver can return a model (concrete input) that will cause that path to be executed. To deal with loops and recursion, a bound is typically enforced on exploration depth. The path constraint updates can be thought of a generating a *symbolic execution tree*, as I illustrate in the following example discussion.

In order to use symbolic execution for side-channel analysis, we must incorporate a model of the side-channel observations into the symbolic program exploration; the path constraints contain only information about how the control flow of the program depends on the symbolic inputs and do not tell us anything about, say, running time. Thus, we augment symbolic execution to maintain a cost model during the symbolic exploration.

The cost model depends on the type of side channel we are interested in analyzing. If we are concerned about memory side channels, we must keep track of calls to memory allocation functions. In my work, I am primarily concerned with timing channels. A very simple model of timing channels is to keep track of the number of executed instructions during execution. Although this is a very course model, it is useful to quantify how much information an attacker can gain in principle. We will see the usefulness of this model in Chapters 4 and 5, which make use of Java Symbolic Path Finder [30] for QIF via symbolic execution. We extend the model further in Chapter 6 to handle more realistic scenarios in which an attacker interacts with the system through a network and must overcome observation noise.

Symbolic Execution Example

In order to give some more intuition about how symbolic execution works, recall the example pseudo-code in Figure 2.2. For the purpose of this example, we will model the side channel observation o using the number of lines of code executed as a proxy for execution time. This makes intuitive sense, as more lines of executed code means longer running time. Although this is clearly not always the case, I temporarily make this simplifying assumption to illustrate the main idea.

In order to perform symbolic execution, we identify symbolic variables H and L to represent the concrete program variables h and l, where h represents a secret PIN number stored in a database, and l represents some guess for the PIN. The reader can follow along with the symbolic execution tree in Figure 2.4. When `checkPIN` is called, the first iteration of the loop is explored. Symbolic execution encounters a branch condition, `if(h[0] != l[0])`, and explores both the true and false branch, as in the first diamond-shaped decision node in the figure. If the condition is true, then the function will return false, the path condition is $\phi_0 \equiv H[0] \neq L[0]$, and $o_0 = 4$ lines of code have been executed.

If the condition is false, then it must be that $H[0] = L[0]$, the next iteration of the loop is executed, where i is incremented to 1, and we compare $H[1]$ and $L[1]$. Again if the condition is true then the function returns false. Symbolic execution determines that the path constraint for this execution path is $\phi_1 \equiv H[0] = L[0] \wedge H[1] \neq L[1]$ and $o_1 = 7$ lines of code have been executed. This continues until the path conditions and complete symbolic execution tree have been generated as shown in Figure 2.4. This serves as a simple example of how symbolic execution is used to automatically explore a program and summarize program behaviors in terms of path constraints. Next I will discuss an extension to symbolic execution that can be used to compute probabilities of program paths.

2.3 Probabilistic Symbolic Execution

Side-channel analysis can be accomplished using information theory (which I will addressed in Section 2.4), which in turn relies on computing probabilities. In the realm of quantitative program analysis, what we seek is the ability to compute the probability of all program paths that lead to the same observations. We will see how this can be used to quantify a program's vulnerability to side-channel attacks in the coming sections. For now, I describe the basic idea behind *probabilistic symbolic execution* [31, 32, 33].

The goal of probabilistic symbolic execution (PSE) is to answer questions of the form: "how likely is a certain program behavior?" or, "what is the the probability of a particular program execution path?" Intuitively, we may reason as follows: the probability that a program execution follows a particular path is equal to the number of inputs that cause that path to be taken, divided by the total possible number of inputs. To formalize this more, first let $\#(\phi_i)$ be the number of solutions to a path constraint ϕ_i and let $\#(D)$ be the size of the program's finite input domain D. Assuming that inputs from D are

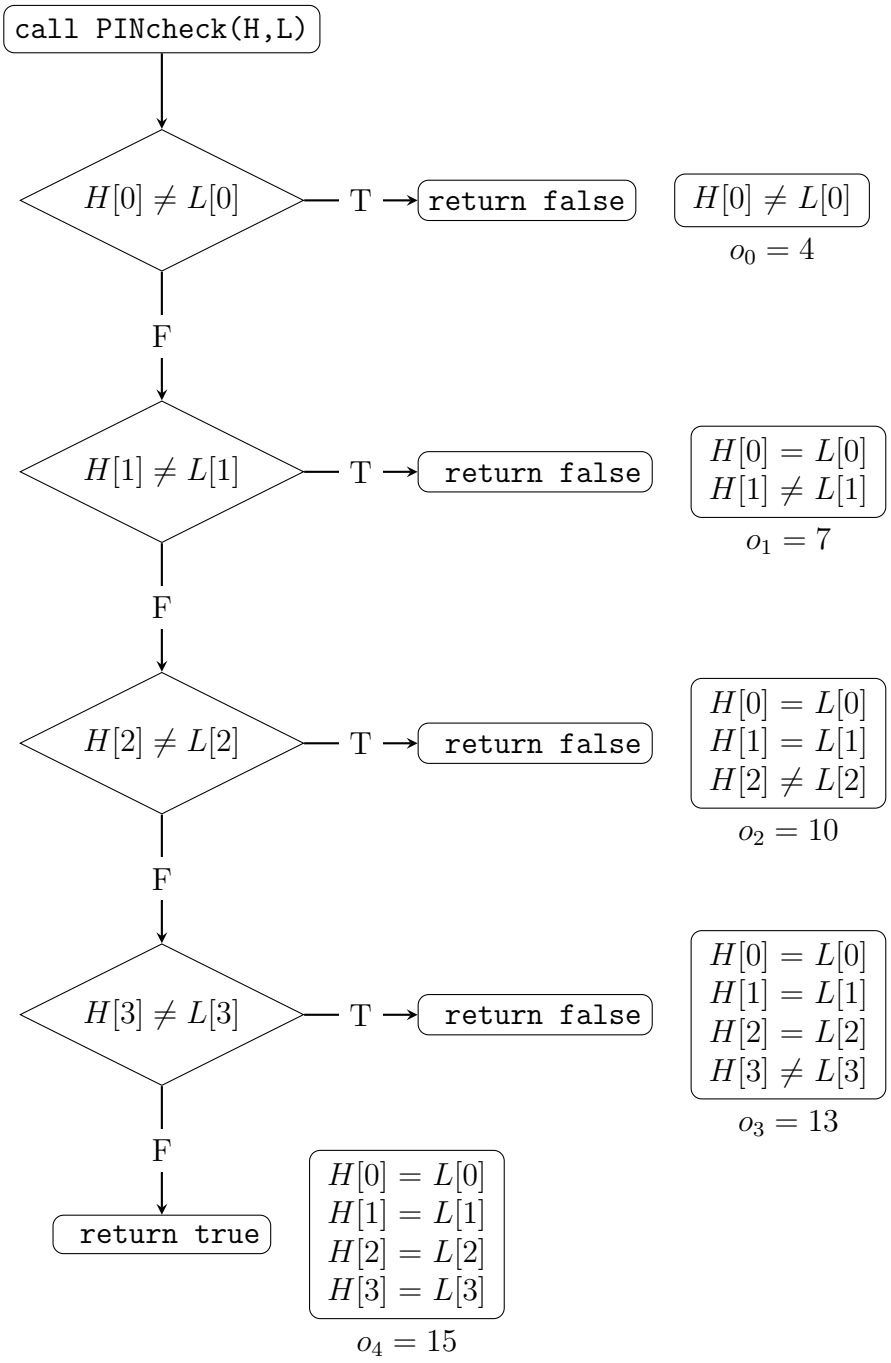


Figure 2.4: Symbolic execution tree and path constraints for the PIN checking code from Figure 2.2.

chosen uniformly at random, we can compute the probability of a path constraint as

$$p(\phi_i) = \frac{\#(\phi_i)}{\#(D)} \tag{2.1}$$

Recall that for the purposes of side-channel analysis, we associate path constraints with a model of side-channel observations. The assumption (which is addressed in detail and subsequently relaxed in Chapter 6), is that inputs that satisfy the same path condition will result in the same observation. Thus, the probability that an attacker can make a particular observation is given by the above-described path-constraint probability, and so for each observation o_i we have that $p(o_i) = p(\phi_i)$. This analysis relies on computing the number of solutions to a path constraint, a problem know as *model counting*. I dedicate Chapter 3 to explaining how to perform model counting.

Probabilistic Symbolic Execution Example

I will continue to use the example of the PIN checking function from Figure 2.2 and corresponding symbolic execution tree from Figure 2.4. I have organized the data from symbolic execution into Table 2.1. We will discuss how the final row, $p(\phi_i)$ is computed.

To make the example simple, suppose that the PIN number H is an array of 4 bits, as well as the attacker's guess L. Thus, the total input domain size is $\#(D) = 2^8 = 256$. We would like to know how many inputs are consistent with each program path. Recall the first path condition: $H[0] \neq L[0]$. There are 8 possible bits which can take on values 0 or 1, and the path constraint restricts one bit to be not equal to another. Thus, there are $2^7 = 128$ possible assignments of bits to H and L. So $p(o_0) = p(\phi_0) = 128/256 = 1/2$. Similar reasoning applies to the remaining path constraints to compute the remaining probabilities. We can consider this table as our first way to quantify the vulnerability of `checkPIN` to side-channel attacks. The probability that an adversary can guess a prefix

i	0	1	2	3	4
ϕ_i	$H[0] \neq L[0]$	$H[0] = L[0]$ $H[1] \neq L[1]$	$H[0] = L[0]$ $H[1] = L[1]$ $H[2] \neq L[2]$	$H[0] = L[0]$ $H[1] = L[1]$ $H[2] = L[2]$ $H[3] \neq L[3]$	$H[0] = L[0]$ $H[1] = L[1]$ $H[2] = L[2]$ $H[3] = L[3]$
return	false	false	false	false	true
$\#(\phi_i)$	128	64	32	16	16
o_i	3	5	7	9	10
$p(\phi_i)$	1/2	1/4	1/8	1/16	1/16

Table 2.1: Each path constraint, ϕ_i, for the code in Figure 2.2, the corrsponding solution count, $\#(\phi_i)$, observation o_i, and observation probability $p(o_i)$.

of the secret PIN of length i in 1 guess is given by p_i. I show in the following two sections how to apply information theory for leakage quantification.

2.4 Quantitative Information Flow

Program analysis methods in the area of *secure information flow* (SIF) track the propagation of sensitive information through a program. SIF detects insecure information flows, commonly known as *information leaks*. These methods produce a binary answer: yes, there is an information leak, or no, there is not, and these methods have seen success in verifying anonymity protocols [34] and firewall protocols [35], and network security protocols [36].

Requiring that a program does not leak *any* information is too strict to be a useful filter for determining program security. The canonical example is that of a password checking function. Each time the password checker rejects an incorrect password, some information about the password is leaked; namely, the number of possible correct passwords is reduced by 1. Indeed, SIF methods will tell us that this program leaks. On the other hand, one can reason that for a sufficiently long password, a brute-force at-

tempt which reduces the search space by 1 with each query to the password checker is an infeasible attack. Hence, contrary to SIF's insecure classification, we would like to say that such a password checking function is in fact secure because the information leakage is small relative to the search space. In a more general setting, the question becomes: given a program, *how much* information is leaked? The ability to answer this question allows us to tolerate small leaks and compare the information leakage of two different implementations. This "how much" question led to the development of *Quantitative Information Flow* (QIF), which gives a foundational framework in which we can measure information leakage [37].

In order to explain how information leakage is quantified, I remind the reader of some terminology and introduce a simple model. We shall consider a program P, which accepts a public low-security input l, a private high security input h, and produces a observation o. In addition, it is customary to introduce the concept of an *adversary*, $\mathcal{A}$. In this model setting, the adversary invokes P with input l and records observation o. $\mathcal{A}$ does not have direct access to h, but would like to learn something about its value. Before invoking P, $\mathcal{A}$ has some initial uncertainty about the value of h, while after observing o, some amount of information is leaked, thereby reducing $\mathcal{A}$'s uncertainty about H. A popular intuitive adage in this setting was popularized by Geoffrey Smith [37]:

"information leaked = initial uncertainty - remaining uncertainty"

The field of QIF formalizes the intuitive statement above by casting the problem in the language of information theory. The field of information theory traces its origins to Claude Shannon's landmark 1948 paper "A Mathematical Theory of Communication" [38], which adapted the concept of *entropy* for the purpose of measuring the amount of information that can be transmitted over a channel, measuring information transmission in *bits of entropy*. In the context of QIF, the information entropy of h is considered a measurement of the adversary's *uncertainty* about h.

I briefly give three relevant information entropy measures [39]. Given a random variable X which can take values in $\{x_1, \ldots, x_n\}$ with probabilities $p(x_i)$, the *information entropy* of X, denoted $\mathcal{H}(X)$ is given by

$$\mathcal{H}(X) = \sum_{x_i \in X} p(x_i) \log_2(1/p(x_i)) \tag{2.2}$$

Given another random variable Y and a conditional probability distribution $p(X|Y)$, we have the *conditional entropy of X given knowledge of Y*:

$$\mathcal{H}(X|Y) = \sum_{y_i \in Y} p(y_i)\mathcal{H}(X|Y = y_i) \tag{2.3}$$

Given these two definitions, the *mutual information of X and Y* is given by

$$\mathcal{I}(X;Y) = \mathcal{H}(X) - \mathcal{H}(X|Y) \tag{2.4}$$

In the context of QIF, we consider random variables H, L, and 0 for the high-security input h, low-security input l, and observation o. We can then interpret, for instance, $p(H)$ to be the adversary's initial belief about H, and the **initial uncertainty** to be $\mathcal{H}(H)$. The conditional entropy $\mathcal{H}(H|O, L)$ quantifies $\mathcal{A}$'s **remaining uncertainty** after providing input L and observing output O. We can then write

$$\mathcal{I}(H;O,L) = \mathcal{H}(H) - \mathcal{H}(H|O,L) \tag{2.5}$$

and interpret $\mathcal{I}(H;O,L)$ as the amount of **information leaked**. These formal definitions are then in line with Smith's intuitive statement of QIF. In addition, if we asssume that the secret and attacker inputs are chosen independently and uniformly at random, we

can make use of well-known indentities of information theory [26] to observe that

$$\mathcal{I}(H; O, L) = \mathcal{H}(H) - \mathcal{H}(H|O, L) = \mathcal{H}(O|H, L) \tag{2.6}$$

Notice that the right hand side of Equation 2.6 is defined in terms of $p(O|H, L)$, and this probability distribution is exactly that which is determined by the path condition probabilities defined in Equation 2.1. For instance, looking at the final row of Table 2.1, we see that the probabilistic symbolic execution table defines a conditional probability distribution of the observation o_i given choices of h and l. Information leakage can be computed from the probabilities that result from symbolic execution and model counting.

I will make extensive use of these concepts of entropy throughout the reset of the dissertation. Later, in Chapter 6, we will require a slightly different formulation of entropy, using the Kullback-Leibler divergence [26]. In the meantime, I will conclude this chapter with an example of how information theory can be used to quantify the information leakage for the two example programs shown earlier.

Side Channel Quantification Example

Using the ideas of probabilistic symbolic execution (Section 2.3) and quantitative information flow (Section 2.4), we can compute the amount of information gained by an attacker for a given program. I illustrate this idea using the running example of Figures 2.2 and 2.4 and Table 2.1.

For the function `checkPIN` we have a set of 5 possible side-channel observations, $\{o_0, o_1, o_2, o_3, o_4, o_5\}$. In addition we have a probability distribution over these observations given by the probabilities $\left\{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{16}\right\}$. We can plug these directly into Equa-

tion 2.6 to compute

$$\mathcal{I}(H; O, L) = \frac{1}{2}\log_2 2 + \frac{1}{4}\log_2 4 + \frac{1}{8}\log_2 8 + \frac{1}{16}\log_2 16 + \frac{1}{16}\log_2 16 = 1.875 \text{ bits}$$

Intuitively this makes sense. Half the time, an attacker will learn the first bit, for which there are 2 possibilities, in which case they gain $\log_2(2) = 1$ bit of information. One quarter of the time, an attacker will learn the first two bits, for which there are 4 possibilities, in which case they gain $\log_2(4) = 2$ bits of information, and so on. Computing the weighted sum of these information gains tells us the amount of information that an attacker can gain on average. What I have illustrated is that this can be computed automatically with symbolic execution, so long as we can compute the number of solutions to a constraint, which I address in the following chapter on model counting techniques.

Now, compare the leakage we just computed to the leakage for the "safe" PIN checking function of Figure 2.3. Since, all executions take the same amount of time, no information can be gained from the side channel. But how much information can be gained from the main channel? The function will return `true` only if all bits match. Since there are 16 possible secrets, this happens with probability $p_T = \frac{1}{16}$, and the function returns `false` with probability $p_F = \frac{15}{16}$. Computing the entropy for this distribution gives us 0.33729 bits of information. Thus, we have a way to quantify the relative vulnerability of two implementations which are functionally equivalent.

2.5 Chapter Summary

In this chapter I introduced the side channel problem and gave some intuition for how it leads to security vulnerabilities. I described how symbolic execution, model counting,

and information theory can be used to automatically quantify side-channel leakage. The rest of this dissertation will explore these ideas in greater detail.

Chapter 3

Model Counting

In this chapter, I discuss the problem of computing the number of solution to a formula, which is known as model counting. As described in Chapter 2, model counting can be used to determine the number of program inputs that satisfy a path condition, which allows us to compute a probability distribution over program side-channel observations and quantify the amount of information an adversary can gain through side-channel attacks. Similar to the way that SMT solvers have been the enabling technology for automated program analysis, model counting is the crucial, enabling technology for automatic quantitative analysis. Many works make use of model counting as the underlying driver behind program analyses and automated reasoning and inference including information flow analysis, execution time estimations, cache analysis, load balancing, reliability analysis, and Bayesian inference [40, 41, 42, 24, 43, 44, 45, 46, 32, 47, 48, 49, 50].

In order to provide background and context for the model counting problem, I will first begin by discussing the classical, historical model counting problem—determining the number of solutions to formulas in propositional logic. I follow that discussion with a description of existing work in model counting for string constraints and then cover work on counting for integer constraints.

In this chapter, I give an overview of those existing techniques before describing my contributions to model counting using automata based methods (Section 3.2.3). My approach to model counting is based on ideas from algebraic graph theory and enumerative combinatorics. Given a deterministic finite automaton that represents all of the solutions to a constraint, I automatically generate a counting function that can be used to compute the number of models of given size.

All of the remaining core chapters of this dissertation utilize model counting as a core component of automated side-channel analysis. We make use of automata-based and polytope-based model counting in Chapter 4 to quantify information leakage for programs with *segment oracle* side channels. In Chapters 5 and 6 I make use of parameterized polytope-based model-counting methods to derive an objective function for synthesizing side-channel attacks.

3.1 Prior Work on Model Counting

In this first section on model counting, I give some background on existing model-counting techniques for propositional logic, strings, and linear integer arithmetic.

3.1.1 Model Counting for Boolean Logic

First, recall the Boolean (propositional logic) satisfiability problem. Given a formula ϕ from propositional logic, is it possible to assign all variables the values T (true) or F (false) so that ϕ evaluates to true? For example, consider the Boolean formula

$$\phi = (x \vee y) \wedge (\neg x \vee z) \wedge (z \vee w) \wedge x \wedge (y \vee v) \tag{3.1}$$

We can observe that ϕ is satisfiable by setting the tuple of variables from ϕ with the

assignment $(x, y, z, w, v) = (T, F, T, F, T)$. In general, a satisfying assignment for ϕ is called a *model* for ϕ. Given a formula ϕ, the model counting problem is to determine how many models are there are for ϕ, and we write $\#(\phi)$ for the model count. Observe that model counting is at least as hard as satisfiability checking, since $|\phi| > 0$ if and only if ϕ is satisfiable.

Exhaustive Enumerative Boolean Model Counting

The most obvious way to count the number of models of a boolean formula is to compute the entire truth table and count the rows which evaluate to true. The truth table for ϕ given in Equation 3.1 is shown in Table 3.1. We can see that there are 6 models for this formula. However, this naive, brute-force method is clearly guaranteed to be exponential in the number of variables. We had to compute $2^5 = 32$ rows in order to count 6 models. Thus, for Boolean formulas, as well as for formulas from other theories, we hope to find methods which do not rely on exhaustive enumerative approaches.

Boolean Model Counting with DPLL

The Davis-Putnam-Logemann-Loveland (DPLL) Algorithm (Algorithm 2) is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulas in conjunctive normal form [59, 60]. First I will give the DPLL algorithm and an example execution. Then I discuss how the DPLL algorithm can be adapted to perform model counting for Boolean formulas.

The DPLL algorithm makes use of a simplification subroutine called unit propagation (Algorithm 1), which I briefly describe. Unit propagation reduces a CNF Boolean formula ϕ to a simpler CNF formula ϕ' that is equisatisfiable with ϕ. A *unit clause* is a clause that is composed of a single literal, u. Since the entire formula must be satisfied, we know that u must be true. In addition, any other clause u' that contains u is automatically

Table 3.1: Complete truth table for ϕ. Rows of the truth table that correspond to models of ϕ are shown in bold.

x	y	z	w	v	$\phi = (x \vee y) \wedge (\neg x \vee z) \wedge (z \vee w) \wedge x \wedge (y \vee v)$
F	F	F	F	F	F
F	F	F	F	T	F
F	F	F	T	F	F
F	F	F	T	T	F
F	F	T	F	F	F
F	F	T	F	T	F
F	F	T	T	F	F
F	F	T	T	T	F
F	T	F	F	F	F
F	T	F	F	T	F
F	T	F	T	F	F
F	T	F	T	T	F
F	T	T	F	F	F
F	T	T	F	T	F
F	T	T	T	F	F
F	T	T	T	T	F
T	F	F	F	F	F
T	F	F	F	T	F
T	F	F	T	F	F
T	F	F	T	T	F
T	F	T	F	F	F
T	**F**	**T**	**F**	**T**	**T**
T	F	T	T	F	F
T	**F**	**T**	**T**	**T**	**T**
T	T	F	F	F	F
T	T	F	F	T	F
T	T	F	T	F	F
T	T	F	T	T	F
T	**T**	**T**	**F**	**F**	**T**
T	**T**	**T**	**F**	**T**	**T**
T	**T**	**T**	**T**	**F**	**T**
T	**T**	**T**	**T**	**T**	**T**

Algorithm 1 Boolean Unit Propagation
Input: CNF formula ϕ, Variable set V, $n = |V|$.
Output: ϕ' equisatisfiable with ϕ

```
1: procedure UnitPropagate(φ, V)
2:     u ← ChooseUnitClause(φ)
3:     delete from φ every clause that contains u (other than u)
4:     delete ¬u from every clause in φ that contains ¬u
5:     repeat until φ does not change
6:     return φ
```

satisfied if u is true, and so u' can be removed. Furthermore, in any clause u'' that contains $\neg u$, the term $\neg u$ cannot contribute to the satisfiability of u'', and so u can safely be deleted from u''. This process can be repeated for all unit clauses until ϕ cannot be further reduced (see Algorithm 1). I demonstrate unit propagation with an example. Recall formula 3.1. Let us rewrite ϕ as a list of conjunctive terms for convenience.

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}$$

Notice that x is a unit clause. So we may remove $x \vee y$. We may also remove $\neg x$ from $\neg x \vee z$ to have

$$\phi = \{z, z \vee w, x, y \vee v\}$$

Now z has become a unit clause, and so we can also remove $z \vee w$. Thus, we have reduced ϕ to a simpler set of clauses that has the same models:

$$\phi = \{z, x, y \vee v\}$$

Notice that ϕ no longer contains the variable w. Unit propagation is able to completely remove variables which do not effect the satisfiability of the formula.

Unit propagation is an optimization for the DPLL Boolean satisfiability procedure.

Algorithm 2 DPLL Boolean Satisfiability
Input: CNF formula ϕ, Variable set V, $|V| = n$.
Output: *true* or *false* (SAT or UNSAT)

```
1: procedure DPLL(φ, V)
2:     φ ← UnitPropagate(φ)
3:     if φ contains a false clause then
4:         return false
5:     else if all clauses of φ are satisfied then
6:         return true
7:     else
8:         x ← SelectBranchVariable(V)
9:         V ← V \ {x}
10:        return DPLL(φ[x ↦ true, V]) ∨ DPLL(φ[x ↦ false, V])
```

DPLL would still work if we did not perform unit propagation, but perhaps more slowly. Regardless, we can now describe the DPLL algorithm (Algorithm 2).

The DPLL algorithm is a search procedure for satisfying assignments for variables of a Boolean formula. DPLL assigns $true$ or $false$ to a variable x and recursively investigates the effect of that assignment on the satisfiability of ϕ. If a recursive branch find a clause that is determined to be $false$, then that branch is terminated and returns $false$. On the other hand, if all clauses of ϕ are satisfied, then that recursive branch determines a a satisfying assignment of the variables in ϕ, and that branch of recursion can return $true$. Otherwise, we combine recursive calls with disjunction so that if any recursive branch returns $true$ then the entire formula is satisfiable. A sample recursion tree of applying the DPLL satisfiability algorithm is shown in Figure 3.1.

Recall that we sought to find a method for counting the number of solutions to ϕ without performing the exhaustive enumerative search, as in Table 3.1. We are in luck, because the DPLL satisfiability procedure can be converted into a procedure for model counting (Algorithm 3) [59]. We add an extra variable t to keep track of how many variables in the formula have not been assigned. If a recursive branch of DPLL finds a clause that is $false$ then that branch of exploration is unsatisfiable and returns 0 as

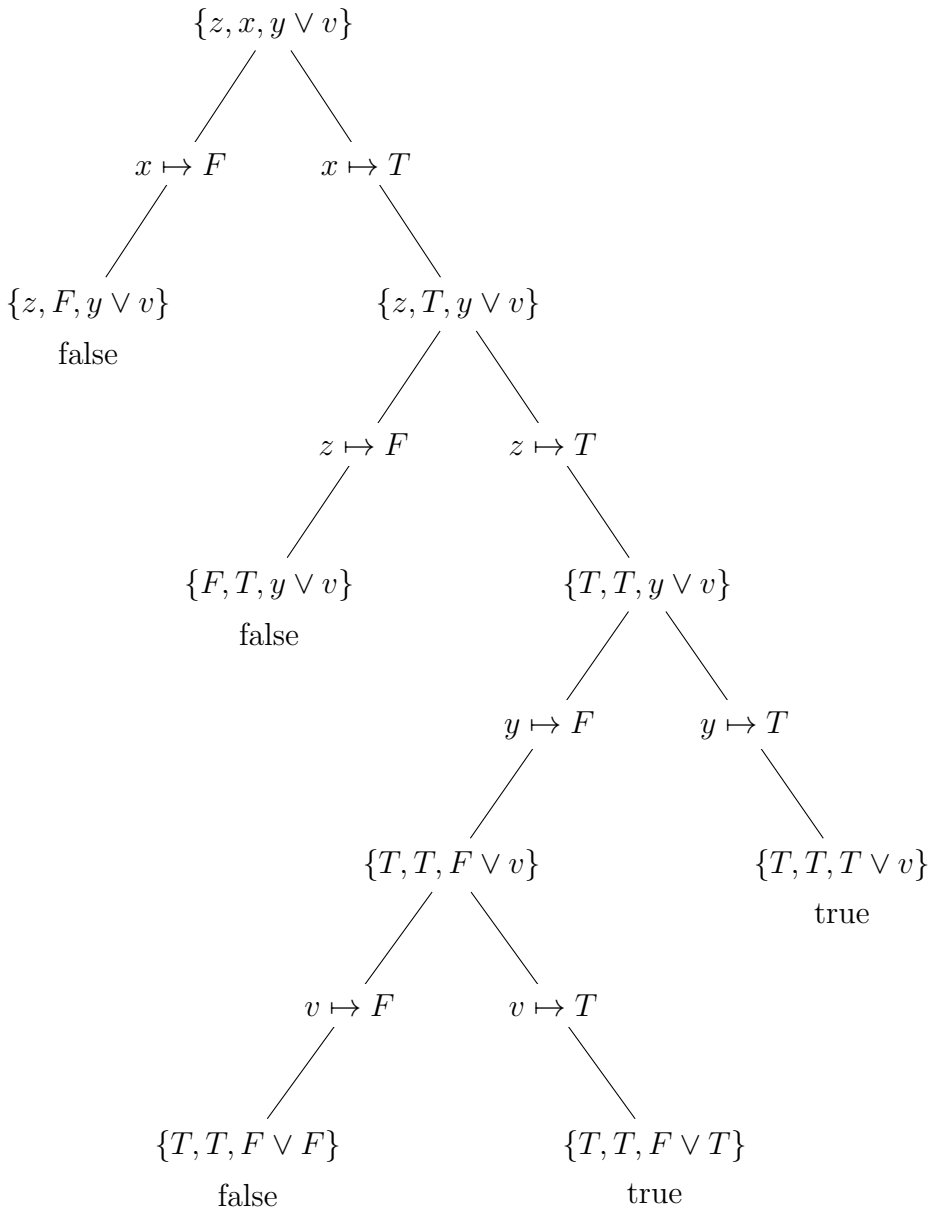


Figure 3.1: Example of DPLL-based satisfiability check for a Boolean formula.

Algorithm 3 DPLL Boolean Model Counting
Input: CNF formula ϕ, Variable set V, # free variables t.
Output: $\#\phi$, the number of models

```
1: procedure DPLL(φ, V, t)
2:     φ ← UnitPropagate(φ)
3:     if φ contains a false clause then
4:         return 0
5:     else if all clauses of φ are satisfied then
6:         return 2^t
7:     else
8:         x ← SelectBranchVariable(V)
9:         V ← V \ {x}
10:        return DPLL(φ[x ↦ true], V, t − 1) + DPLL(φ[x ↦ false], V, t − 1)
```

the count for that branch. If instead, all clauses are satisfied, then that branch returns 2^t, as each unassigned variable can take on two possible values. Otherwise, as in the DPLL satisfiability checking algorithm, we recursively explore both branches. Again, we recursively investigate assigning $true$ or $false$ to a variable of the formula. When we make a recursive call after assigning $true$ or $false$, we decrement t. The number of models is then the sum of the number of models returned by each recursive call (whereas in the original DPLL algorithm we used disjunction to combine recursive calls). A sample recursion tree of applying the DPLL satisfiability algorithm is shown in Figure 3.2, which tells us that the number of models is 6 by summing the values found in the leaves. We get the same result as in the naive, brute-force approach, but without enumerating all possible variable assignments.

Now that we have discussed how to perform model counting for Boolean formulas, we move on to more interesting theories: first strings and then integers.

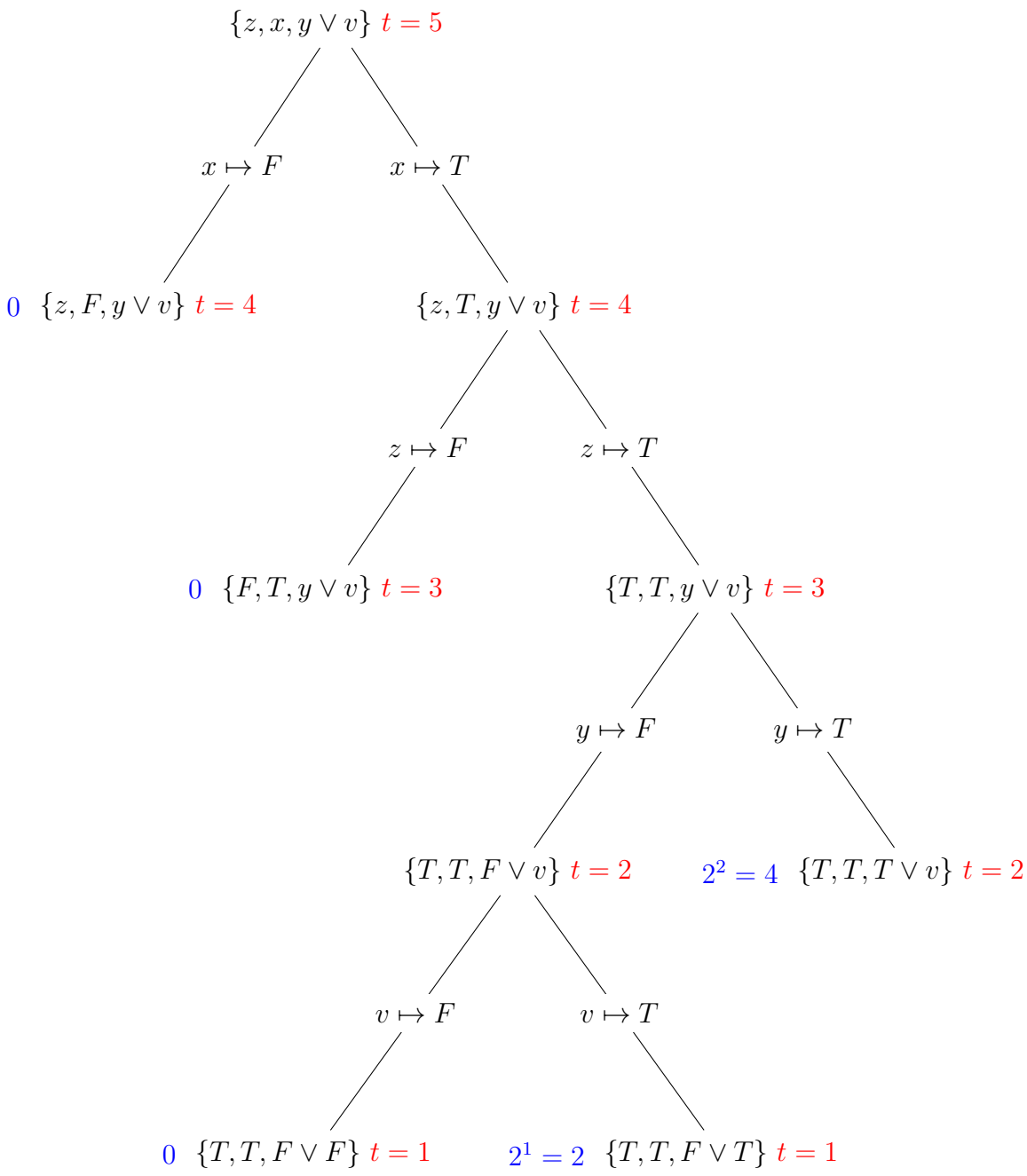


Figure 3.2: Example of DPLL-based model counting for a Boolean formula.

3.1.2 Model Counting for String Constraints

We now turn our attention to counting models for constraints over variables of the string type. The amount of string-manipulating code in modern software applications has been increasing. Common uses of string manipulation include: 1) Input sanitization and validation in web applications; 2) Query generation for back-end databases; 3) Generation of data formats such as XML and HTML; 4) Dynamic code generation; 5) Dynamic class loading and method invocation. Due to the growing proliferation of web applications that make heavy use of string manipulation, there is a growing body of work on string analysis [51, 52, 53, 54, 55]; however none of these earlier approaches provide model-counting functionality. In more recent times, due to the importance of model counting in quantitative program analyses, model counting constraint solvers are gaining increasing attention. String Model Counter (SMC) [44] is one existing tool for model counting of string constraints, which we will describe later in this section.

First, in order to motivate this discussion we begin with a simple problem, assuming that the reader is familiar with regular expressions. One possible simple type of constraint on a string variable is that it is a member of a regular expression made of constant characters, concatenations (written here as juxtaposition), alternation (+), and Kleene closure (*). Consider a regular expression constraint over the alphabet $\Sigma = \{0, 1\}$:

$$X \in (0|(1(01^*0)^*1))^*$$

We can now ask ourselves, "how many models are there for X?", where a model is an assignment of a string value to X that is in the regular expression. Well, that's easy; since the regular expression contains the Kleene closure operator, there are infinitely many solutions for X. That is not a very satisfying answer. Let us refine the question we are asking so that we get a more informative answer. Suppose that we want to know,

Length, k	Strings, X	Count, a_k
0	ε	1
1	0	1
2	11	1
3	110	1
4	$1001, 1100, 1111$	3
5	$10010, 10101, 11000, 11011, 11110$	5

Table 3.2: Model counting table for strings X.

for a given integer K, how many solutions are there for X, of length k?

As in the case of Boolean model counting, we can attempt to begin naively by brute-force enumeration. First, ε is a model, and it is the only model of string length 0. The string 0 is also a model, and it is the only model of length 1. Going further, 11 and 110 are the only models of lengths 2 and 3 respectively. Finally, for length 4 there are 3 different strings that satisfy the constraint: $1001, 1100$, and 1111 are all models of length 4. It turns out that there are 5 models of length 5. Like we said, there are infinitely many models, so we won't continue listing them, bet we can organize our results so far in a table. If we let a_k represent the number of strings of length k then we can see some partial counting results in Table 3.2.

Clearly, for any constraint C, we can generate all strings of length k and check if they satisfy C, count how many there are, and build up a counting table. But, just as in the Boolean model counting problem, we would like a solution that is better than naive brute-force enumeration, which tells us the number of strings of length k for any k. We call this *string model counting parameterized by length*. I will discuss two methods for solving this problem. The first is based on a set of syntactic transformations and the second, discussed in the following section, is based on constructing deterministic finite automata (DFA) that represent the set of solutions to a string constraint. Both methods make use of generating functions which I now cover briefly.

Generating Functions

Generating functions are a useful mathematical tool for counting combinatorial objects [61, 62, 63]. Given any (possibly infinite) sequence with integer indices, $\{a_i\} = a_1, a_2, a_3, \ldots$, the *generating function* for the sequence is a polynomial $g(x)$ where the coefficient of x^i is a_i. We sometimes say that $g(x)$ *encodes* the sequence and we write

$$g(x) = \sum_{i=0}^{\infty} a_i x^i = a_o + a_1 x + a_2 x^2 + a_3 x^3 + \ldots \tag{3.2}$$

Generating functions are useful for string counting because generating functions can often be written in a compact form as a rational expression. Recall, the Taylor series expansion formula to expand f about the point $x = 0$:

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!} x^n \tag{3.3}$$

Using power series expansions, $g(x)$ can be written as a ratio of polynomials p and q such that the Taylor expansion is equal to g.

$$g(x) = \sum_{i=0}^{\infty} a_i x^i = \frac{p(x)}{q(x)} \tag{3.4}$$

Generating functions written as a ratio of polynomials can be systematically composed to recursively encode counting sequences for subexpressions of string constraints.

Examples of Generating Functions Let us give some simple examples of generating functions.

Example 1. Constant sequence of 1's. Consider the infinite sequence where every $a_i = 1$: $\{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, \ldots\}$. The generating function for this sequence is

$$g(x) = 1 + 1x + 1x^2 + 1x^3 + 1x^4 + 1x^5 + 1x^6 + 1x^7 + 1x^8 + 1x^9 + 1x^{10} + \ldots$$

This is an infinite polynomial. However, consider the rational expression $g(x) = \frac{1}{1-x}$, and compute the power series expansion of $g'(x)$ about $x = 0$ using Equation 3.8:

$$g(x) = 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} + \ldots$$

So, $g(x) = \frac{1}{1-x}$ is an equivalent expression that encodes the infinite sequence of 1's by taking the series expansion.

Example 2. Non-negative integers. Consider the infinite sequence where $a_i = i$: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots\}$. Then the generating function for this is an infinite polynomial which has the finite rational representation shown here:

$$g(x) = x + 2x^2 + 3x^3 + 4x^4 + 5x^5 + 6x^6 + 7x^7 + 8x^8 + 9x^9 + 10x^{10} + \ldots = \frac{x}{(x-1)^2}$$

Example 3. Alternating +/- 1. The sequence of alternating positive and negative 1's has the generating function:

$$g(x) = \frac{1}{x+1} = 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 - x^9 + x^{10} + \ldots$$

Example 4. Powers of base b. Consider the sequence of non-negative integer powers of a number b, where $a_i = b^i$: $\{1, b, b^2, b^3, b^4, b^5, b^6, b^7, b^8, b^9, b^{10}\}$. The generating function for this sequence is:

$$g(x) = \frac{1}{1-bx} = 1 + bx + b^2x^2 + b^3x^3 + b^4x^4 + b^5x^5 + b^6x^6 + b^7x^7 + b^8x^8 + b^9x^9 + b^{10}x^{10} + \ldots$$

Example 5. Binary strings of length i. Let a_i be the number of binary strings of length i. That is, $a_i = |\{s : s \in (0+1)^* \wedge \text{length}(s) = i\}|$. From our familiarity with binary numerals, we know that the i^{th} coefficient of the generating function should be 2^i. In fact, this is a special case of the previous example with $b = 2$. Thus, the generating function we seek is

$$g(x) = \frac{1}{1-2x}$$

Syntax-Based Generating Functions for String Counting

Given a constraint C on a string variable X, let $\mathcal{L}$ be the language of strings that satisfy C. We would like a way to construct a generating function that encodes the number of strings of length k in $\mathcal{L}$. More specifically, a counting sequence for language $\mathcal{L}$ encodes

$$a_k = |\{x : x \in \mathcal{L} \wedge \text{len}(x) = k\}|$$

Recall the example constraint $C \equiv X \in (0|(1(01^*0)^*1))^*$ from the introduction of this section. We constructed the first 5 terms of the counting sequence by brute-force enumeration in Table 3.2. So, the corresponding counting sequence is

$$a_0 = 1, a_1 = 1, a_2 = 1, a_3 = 1, a_4 = 3, a_5 = 5, \ldots$$

From our discussion in the previous section we know that we can encode this sequence

Algorithm 4 Regular Expressions Generating Function Construction

```
1: procedure GeneratingFunction(e)
2:     e match
3:         case ε
4:             return 1
5:         case c
6:             return z
7:         case e1|e2
8:             return GeneratingFunction(e1) + GeneratingFunction(e2)
9:         case e1 ∘ e2
10:            return GeneratingFunction(e1) × GeneratingFunction(e2)
11:        case e1*
12:            return 1/(1 − GeneratingFunction(e1))
```

as a generating function that starts off with coefficients corresponding to those 5 terms:

$$g(z) = 1 + x + +x^2 + x^3 + 3x^4 + 5x^5 + \dots$$

But how do we compute the remaining terms? Or, how do we compute a generating function that encodes the entire infinite sequence? This problem is addressed in a classic work of combinatorial properties of formal languages by Chomsky and Schützenberger [63]. Recent work has shown how to solve this problem for more complex string constraints, and a tool[1] called "String Model Counter" was implemented [44]. Here we give the major idea behind these works.

The main insight for constructing a counting generating function for an unambiguous regular expression e is to recursively compute generating function for subexpressions of e and combine them in the appropriate way, using purely syntactic transformations (see Algorithm 4).

- The simplest transformation is for the empty string; ε is the only string of length

[1]SMC Online: `https://github.com/loiluu/smc`

0, and so the generating function should be $1z^0 = 1$.

- For any constant single character c, there is one corresponding string of length 1 (namely c), and so the generating function is $1z^1 = z$.

- When we compute $e = e_1|e_2$, the resulting number of strings of length k in e is the sum of the numbers of strings of length k in each of e_1 and e_2, since e was assumes to be unambiguous. Thus, the generating function for $e_1|e_2$ is the sum of the generating functions for e_1 and e_2.

- When we compute $e = e_1 \circ e_2$, the resulting number of strings of length k in e is the product of the numbers of strings of length k in each of e_1 and e_2, again since e was assumes to be unambiguous. Thus, the generating function for $e_1 \circ e_2$ is the product of the generating functions for e_1 and e_2.

- The final rule says that if $e = e_1^*$ then the generating function is$1/(1 - GF(e_1))$. This rule can be seen as the generating function for the sequence $a_i = m^i$, for some m, as the i-th term of the series expansion of $1/(1 - mx)$ is $m^i x^i$. Thus, if e_1 corresponds to an expression for m strings, then the expression encodes the number of strings of all possible finite concatenations of strings from e_1. In order to gain more intuition on this rule, one can review examples 1, 4, and 5 in the earlier discussion of generating functions.

We apply the recursive generating function procedure to the example constraint $C \equiv X \in (0|(1(01^*0)^*1))^*$. The parse tree of the constraint is shown in Figure 3.3. This tree also corresponds to the recursion tree that results from executing the generating function construction algorithm of Algorithm 4 on C. One can see that the generating function is computed bottom-up from the leaves of the parse tree, with the final generating function at the root of the tree, and shown here:

$$g(z) = \frac{1}{1 - z - \frac{z^2}{1 - \frac{z^2}{1-z}}} = \frac{1 - z - z^2}{(z-1)(2z^2 + z - 1)}$$

If we compute the series expansion of $g(z)$ we can determine the number of strings of any length k by computing the kth series coefficient.

$$g(z) = 1z^0 + 1z^1 + 1z^2 + 1z^3 + 3z^4 + 5z^5 + \ldots$$

I described how syntax-based transformation rules can be used to construct a generating function for counting the number of solutions to a string constraint that consists of regular language membership. However, string constraints which come from programs often have much more complex expressions like $x = y.\texttt{substring}(3, 10) \wedge x.\texttt{charat}(4) = a$. The authors of SMC provide more complex syntactic transformations for constraint operators like `substring`, `charat`, `indexof`, `replace`, and so on [44]. These remaining syntax-based generating function rules are beyond the scope of this dissertation and so I do not discuss them further.

One drawback to the syntax-based transformations is that after a transformation is applied, the semantics of the constraint are "forgotten". This leads to problems in both overcounting and undercounting solutions. The purely syntactic methods of SMC cannot propagate string values across logical connectives which reduces its precision. SMC addresses this by providing rules which give generating functions for both an upper and lower bound for the count. For example, for a simple constraint such as $(x \in a|b) \vee (x \in a|b|c|d)$ SMC will generate a model-count range which consists of an upper bound of 6 and a lower bound of 2, whereas the exact count is 4. The essential problem here is that of ambiguity. If a string can be generated in more than one way, then that string will be counted more than once. Moreover, SMC always generates a lower

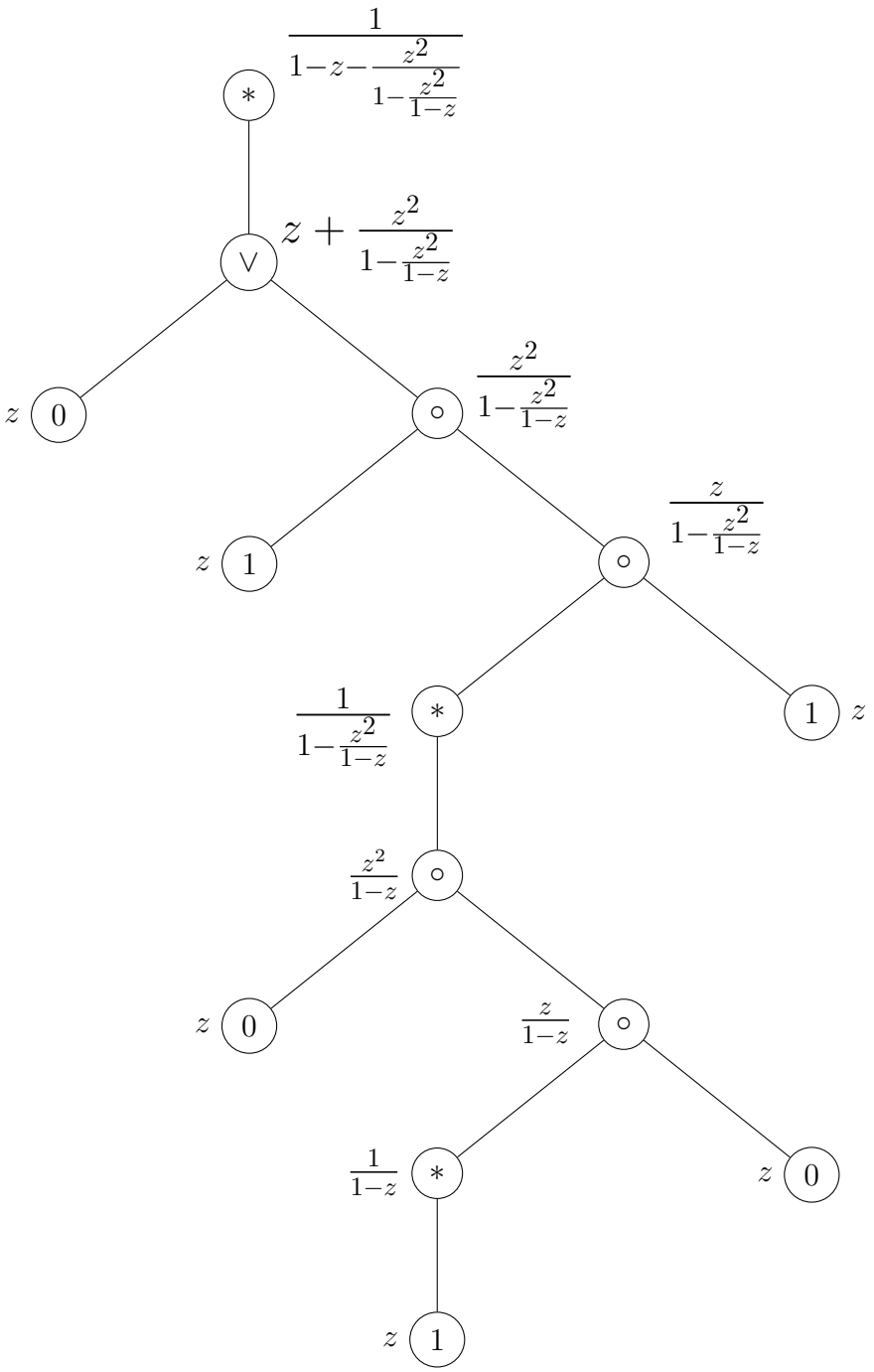


Figure 3.3: Parse tree for example regular expression to illustrate recursive construction of generating function from the leaves up according to the rules in Figure 3.3.

bound of 0 for conjunctions that involve the same variable. So, the range generated for $(x \in a|b) \ \wedge \ (x \in a|b|c|d)$ would be 0 to 2, whereas the exact count is 2. In the next section I will discuss automata-based model counting methods which retain the semantic content of a string constraint, allowing for more precise model counting.

3.1.3 Linear Integer Arithmetic

I now discuss methods for counting the number of solutions to constraints from linear integer arithmetic (LIA). One approach to model counting for constraints from the theory of linear integer arithmetic is to consider the set of solutions to be the integer lattice points of $\mathbb{Z}^n$ located in the interior of a polytope in $\mathbb{R}^n$. These techniques rely on methods for integrating over discrete domains and generating functions. While the specific details for the inner workings of these methods is beyond the scope of this dissertation, I discuss two specific linear integer arithmetic model counting tools and give a high level idea of what they compute.

LattE (Lattice Enumeration)

LattE determines the number of tuples of points that satisfy a system of linear inequalities of the form $A\mathbf{x} \leq \mathbf{b}$, where $A_{m\times n}$ is a matrix of integer coefficients, $\mathbf{b}_{n\times 1} \in \mathbb{Z}^n$ is a constant vector, and $\mathbf{x}_{n\times 1} \in \mathbb{Z}^n$ is the vector of unknown integer values [56, 64]. LattE implements the polynomial time Barvinok algorithm to return the model count, and is used primarily to count points inside bounded concrete regions. I give an example to illustrate this idea and contrast with counting for counting inside unbounded symbolic regions in the following discussion about the Barvinok Model counter. Consider the constraint

$$x \geq 0 \ \wedge \ y \geq 0 \ \wedge \ y \leq x \ \wedge \ 2y \leq 6 \ \wedge \ x \leq 6$$

We can write this as a system of inequalities in the desired $A\mathbf{x} \leq \mathbf{b}$ form as

$$\begin{pmatrix} -1 & 0 \\ 0 & -1 \\ -1 & 1 \\ 0 & 2 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 6 \\ 6 \end{pmatrix}$$

We can visualize the solution space as the integer lattice points contained in the intersection of half-planes defined by the x-axis, y-axis, and lines l_1, l_2, and l_3 defined by the system of inequalities (Figure 3.4), including the boundaries. We can see that the model count is 22 by counting the contained lattice points. LattE computes model counts for systems of linear inequalities, in an arbitrary dimension using polytope decompositions and generating functions using the Barvinok polynomial time algorithm.

Barvinok

Similar to LattE, Barvinok is a model counter for linear integer arithmetic constraints, but with more functionality. Barvinok can generate a symbolic model-counting function for symbolically defined polytopes as well as perform *weighted* model counting.

The previous discussions have implicitly assumed that we are interested in unweighted model counting, i.e., the problem of counting the number of models of a formula where each model has weight equal to 1. If one assigns an arbitrary weight to each model, one may compute the sum of the weights of all models. Later in this dissertation I am interested performing side-channel analysis in which the distribution of the secret input

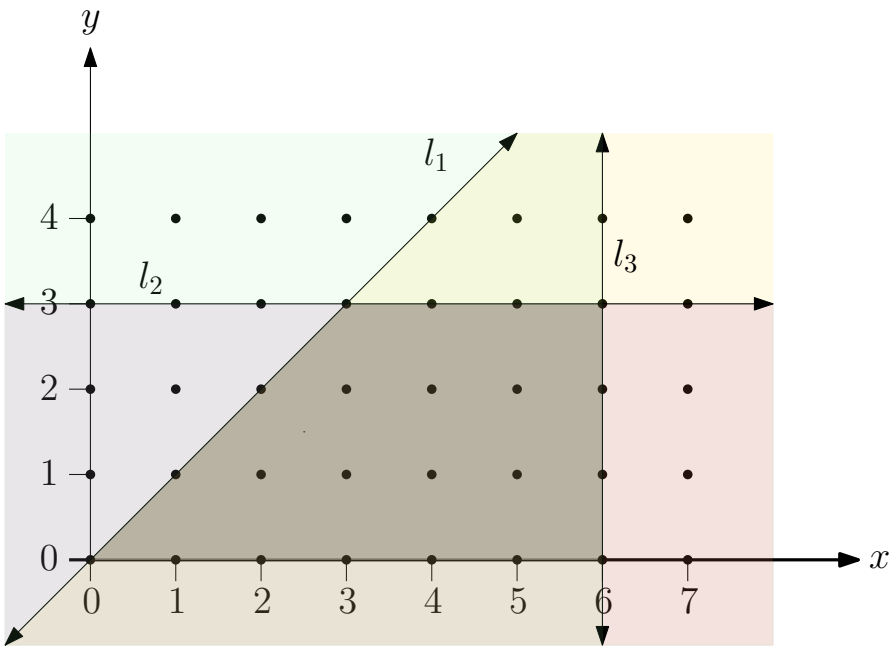


Figure 3.4: Satisfying solutions for the example constraint are the integer lattice points contained in the trapezoidal area. There are 22 points.

h is non-uniform. I accomplish this by counting models for $\phi(h, l)$ where the weight of each model $p(H = h)$. I make use of symbolic and weighted-symbolic model counting in Chapters 5 and 6.

One may compute weighted model counts by summing over all possible h, but this would be inefficient if the domain of h is very large. Hence, we seek an efficient way to compute the weighted model count. This is accomplished by using symbolic weighted model counting. I first give an example.

For now we will assume the unweighted version of the problem, or equivalently, every model has weight 1. Now, recall the constraint that we previously discussed, but where we replace the final conjunct $x \leq 6$ with $x \leq t$, where t is an integer parameter.

$$x \geq 0 \ \wedge \ y \geq 0 \ \wedge \ y \leq x \ \wedge \ 2y \leq 6 \ \wedge \ x \leq t$$

Again, we can write this as a system of inequalities in the desired $A\mathbf{x} \leq \mathbf{b}$ form, where

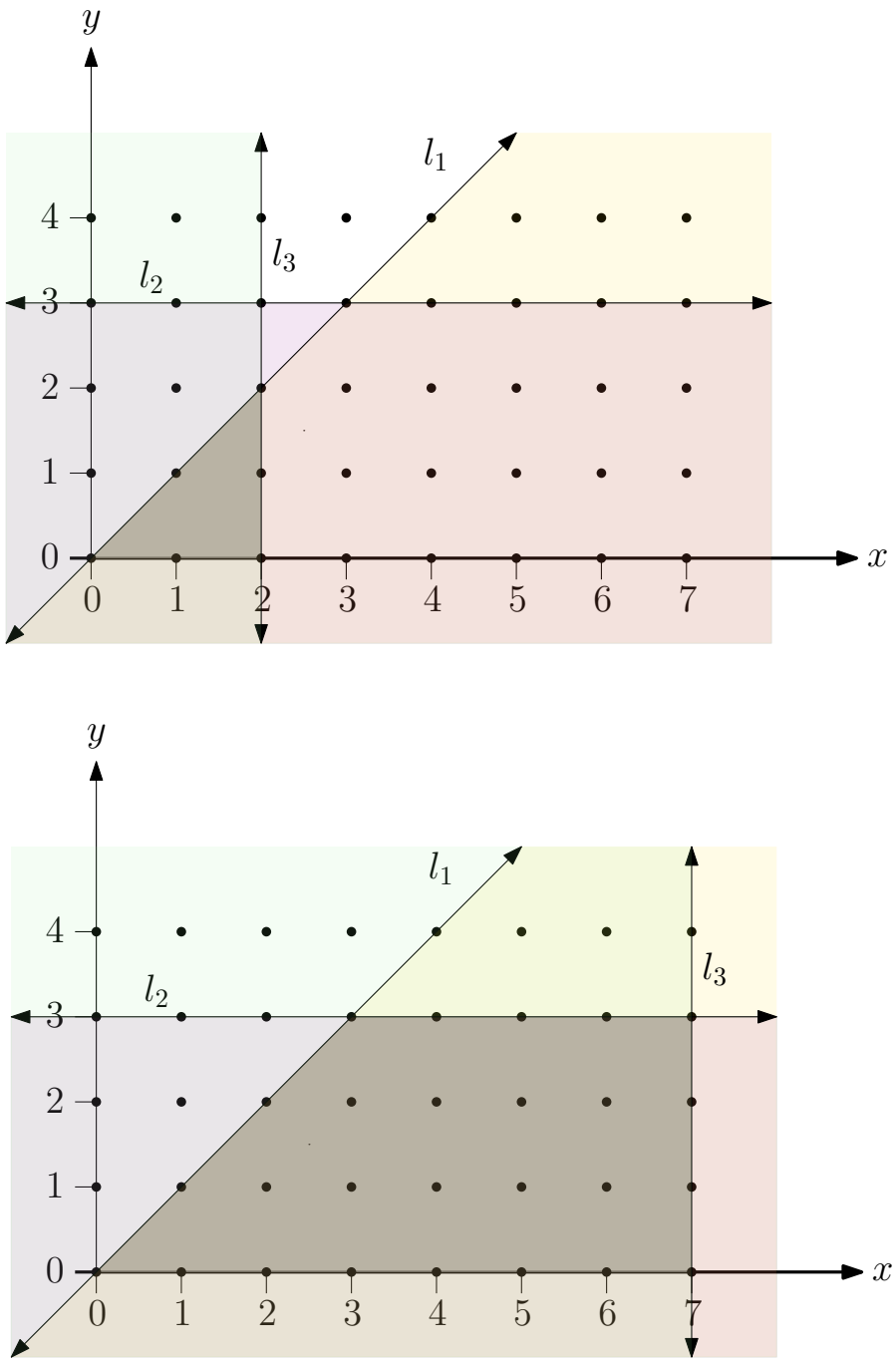


Figure 3.5: Satisfying solutions for the example constraint parameterized by t. For $t = 2$ (top) there are 6 models and for $t = 7$ (bottom) there are 26 models.

t is included as a symbolic parameter. If we set $t = 2$ then the model count is 6 (top figure) and if we set $t = 7$ the model count is 26.

$$\begin{pmatrix} -1 & 0 \\ 0 & -1 \\ -1 & 1 \\ 0 & 2 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 6 \\ t \end{pmatrix}$$

Now, different values of t result in different intersecting half-planes, thereby resulting in different model counts, depending on the choice of t. This is what we mean by symbolic model counting. Consider Figure 3.5, which illustrates the solution space for two different values of t. For $t = 2$ (top) there are 6 models and for $t = 7$ (bottom) there are 26 models. If we use Barvinok to perform model counting, it tells us that the number of satisfying solutions for points (x, y) is a piecewise polynomial function of t:

$$f(t) = \begin{cases} \frac{1}{2}t^2 + t + 1 & 0 \leq t \leq 3 \\ 10 + 4t & 0 \leq t > 3 \\ 0 & \text{otherwise} \end{cases}$$

Thus, we can evaluate the model count for any value of t that we want without making further calls to the model counter. However, consider the weighted version where each point (x, y) has a weight $w(x, y)$. For example, suppose that $w(x, y) = y$. That is, the weight of a point is given by the y-coordinate. Then Barvinok will tell us that the weighted model counting function parameterized by t is given by

$$f(t) = \begin{cases} \frac{1}{6}t^3 + \frac{1}{2}t^2 + \frac{1}{3}t & 0 \leq t \leq 3 \\ 6t - 8 & 0 \leq t > 3 \\ 0 & \text{otherwise} \end{cases}$$

Barvinok performs weighted model counting by representing a linear integer arithmetic constraint ϕ on variables $X = \{x_1, \ldots, x_n\}$ with weight function $W(X)$ as a symbolic polytope $Q \subseteq \mathbb{R}^n$. Let $Y \subseteq X$ be a set of parameterization variables and Y' be the remaining free variables of X. Barvinok's polynomial-time algorithm generates a (multivariate) piecewise polynomial F such that $F(Y)$ evaluates to the weighted count of the assignments of integer values to Y' that lie in the interior of Q. For side-channel analysis, we are interested in computing the probability of an observation given a choice of l and the probability distribution of high security inputs. Thus, we let Y be the set of low security variables, Y by the set of high security variables, and W be $p(H = h)$.

3.2 Automata-Based Model Counting

In the previous section we described prior, existing work on model counting for propositional logic, string constraints, and linear integer arithmetic constraints. In this section I describe our work which gives a method for model counting for a strings constraints. Given a string constraint, C, this method constructs a deterministic finite automata (DFA) that represents all solutions to the constraint C. Then, determining the number of solutions to the constraint reduces to counting the number of accepting paths in the DFA. This is accomplished using algebraic graph theory and generating functions.

3.2.1 String Constraints

We handle string constraints from the following grammar, where C denotes the basic constraints, n denotes integer values, $s \in \Sigma^*$ denotes string values, ε is the empty string, v denotes string variables, $\circ$ is the string concatenation operator, $\textsc{len}(v)$ denotes the length of the string value that is assigned to variable v.

$$
\begin{array}{lcl}
F & \rightarrow & C \mid \neg F \mid F \ \wedge \ F \mid F \ \vee \ F \\
C & \rightarrow & S \in R \\
 & \mid & S = S \\
 & \mid & S = S \ . \ S \\
 & \mid & \textsc{len}(S) \ O \ n \\
 & \mid & \textsc{len}(S) \ O \ \textsc{len}(S) \\
 & \mid & \textsc{contains}(S, s) \\
 & \mid & \textsc{begins}(S, s) \\
 & \mid & \textsc{ends}(S, s) \\
S & \rightarrow & v \mid s \\
R & \rightarrow & s \mid \varepsilon \mid R \circ R \mid R \shortmid R \mid R^* \\
O & \rightarrow & < \ \mid \ = \ \mid \ >
\end{array}
$$

3.2.2 Automata Construction

I now describe the automata construction algorithm. A Deterministic Finite Automaton (DFA) A is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where $Q = \{1, 2, \ldots, n\}$ is the set of n states, Σ is the input alphabet, $\delta \subseteq Q \times Q \times \Sigma$ is the state transition relation set, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final, or accepting, states.

Given an automaton A, let $\mathcal{L}(A)$ denote the set of strings accepted by A. Given a constraint F and a string variable v, our goal is to construct a deterministic finite automaton (DFA) A, such that $\mathcal{L}(A) = [\![F, v]\!]$ where $[\![F, v]\!]$ denotes the set of strings for which F evaluates to true when substituted for the variable v in F.

Let us define an automata constructor function $\mathcal{A}$ such that, given a string constraint

F and a variable v, $\mathcal{A}(F, v)$ is an automaton where $\mathcal{L}(\mathcal{A}(F, v)) = [\![F, v]\!]$. Below we discuss how to implement the automata constructor function $\mathcal{A}$.

Let $\mathcal{A}(\Sigma^*), \mathcal{A}(\Sigma^n), \mathcal{A}(s)$, and $\mathcal{A}(\emptyset)$ denote automata that accept the languages Σ^*, Σ^n, $\{s\}$ (constant string s), and $\emptyset$, respectively. We construct the automaton $\mathcal{A}(F, v)$ recursively on the structure of the single-variable constraint F. We assume the standard DFA algorithms for the operations complement, union, intersection, concatenation, and constructing a DFA from a regular expression (refer to Algorithm 5). Boolean operations for formulas, $\neg$, $\vee$, and $\wedge$ are handled with DFA complement, union, and intersection (lines 7 - 12). Regular expression constraints are handled using the DFA construction algorithm for regular expressions (lines 13 - 14). Equality with a constant string is handled by constructing a DFA that accepts only that string (lines 15 - 16). For constraints on the length of a string variable we construct DFAs that accept strings according to that length restriction (lines 17 - 24). For constraints that specify that v begins, ends, or contains a constant string s, we construct the automata that concatenates $\mathcal{A}(s)$ with Σ^* in the relevant positions (lines 25-30).

Example DFA Construction

As a simple example of how DFA are recursively constructed, consider the string constraint $F \equiv \neg(x \in (01)^*) \wedge \textsc{len}(x) \geq 1$ over the alphabet $\Sigma = \{0, 1\}$. Let us name the sub-constraints of F as $C_1 \equiv x \in (01)^*$, $C_2 \equiv \textsc{len}(x) \geq 1$, $F_1 \equiv \neg C_1$, where $F \equiv F_1 \wedge C_2$. Figures 3.7 and 3.6 demonstrate the automata construction algorithm on our running example. The algorithm starts from the constraints at the leaves of the syntax tree (C_1 and C_2), and constructs automata for them. Then it traverses the syntax tree towards the root by constructing an automaton for each node using the automata constructed for its children. The automaton for F_1 is constructed using the automaton for C_1. The solution DFA for F is constructed using the automata for F_1 and C_2.

Algorithm 5 DFA Construction Procedure for Constraint F

```
1: procedure 𝒜(F)
2:     F match
3:         case true
4:           return 𝒜(Σ*)
5:         case false
6:           return 𝒜(∅)
7:         case ¬F′
8:           return DFAComplement(𝒜(F))
9:         case F1 ∨ F2
10:          return DFAUnion(𝒜(F1), 𝒜(F2))
11:        case F1 ∧ F2
12:          return DFAIntersect(𝒜(F1), 𝒜(F2))
13:        case v ∈ R
14:          return DFARegEx(R)
15:        case v = s
16:          return 𝒜(s)
17:        case len(v) = n
18:          return 𝒜(Σ^n)
19:        case len(v) < 1
20:          return 𝒜(Σ)
21:        case len(v) < n
22:          return DFAUnion(𝒜(Σ^n), 𝒜(len(v) < n − 1))
23:        case len(v) > n
24:          return DFAConcatenate(𝒜(Σ^{n+1}), 𝒜(Σ*))
25:        case contains(v, s)
26:          return DFAConcatenate(𝒜(Σ*), 𝒜(s), 𝒜(Σ*))
27:        case begins(v, s)
28:          return DFAConcatenate(𝒜(s), 𝒜(Σ*))
29:        case ends(v, s)
30:          return DFAConcatenate(𝒜(Σ*), 𝒜(s))
```

3.2.3 Model Counting with Automata

Once we have translated a set of constraints into an automaton we employ algebraic graph theory [66] and analytic combinatorics [62] to perform model counting. In our method, model counting corresponds exactly to counting the accepting paths of the

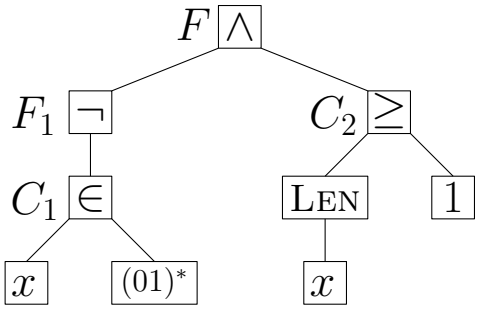


Figure 3.6: The syntax tree for the string constraint $\neg(x \in (01)^*) \ \wedge \ \text{LEN}(x) \geq 1$.

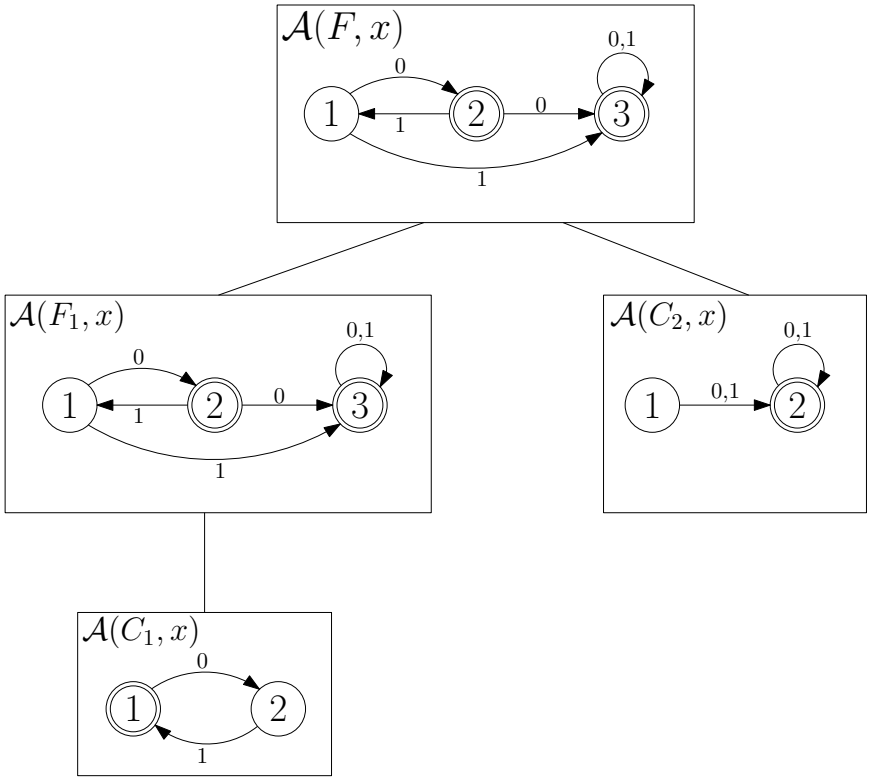


Figure 3.7: The automata construction that traverses the syntax tree of Figure 3.6 from the leaves upward. The solution DFA is at the root.

constraint DFA up to a given length bound k. This problem can be solved using dynamic programming techniques in $O(k \cdot |\delta|)$ time where δ is the DFA transition relation [67, 68]. However, for each different bound, the dynamic programming technique requires another traversal of the DFA graph.

A preferable solution is to derive a symbolic function that given a length bound k

outputs the number of solutions within bound k. To achieve this, we use the *transfer matrix method* [61, 62] to produce an ordinary generating function which in turn yields a linear recurrence relation that is used to count constraint solutions. We will briefly review the necessary background and then describe the model counting algorithm.

Given a DFA A, consider its corresponding language $\mathcal{L}$. Let $\mathcal{L}_i = \{w \in \mathcal{L} : |w| = i\}$, the language of strings in $\mathcal{L}$ with length i. Then $\mathcal{L} = \bigcup_{i \geq 0} \mathcal{L}_i$. Define $|\mathcal{L}_i|$ to be the cardinality of $\mathcal{L}_i$. The cardinality of $\mathcal{L}$ can be computed by the sum of a series $a_0, a_1, \ldots, a_i, \ldots$ where each a_i is the cardinality of the corresponding language $\mathcal{L}_i$, i.e., $a_i = |\mathcal{L}_i|$.

For example, recall the automaton in Fig. 1. Let $\mathcal{L}^x$ be the language over $\Sigma = \{0, 1\}$ that satisfies the formula $(x \notin (01)^* \wedge \text{LEN}(x) \geq 1)$. Then $\mathcal{L}^x$ is described by the expression $\Sigma^* - (01)^*$. In the language $\mathcal{L}^x$, we have zero strings of length 0 ($\varepsilon \notin \mathcal{L}^x$), two strings of length 1 ($\{0, 1\}$), three strings of length 3 ($\{00, 10, 11\}$), and so on. The sequence is then $a_0 = 0, a_1 = 2, a_2 = 3, a_3 = 8, a_4 = 15$, etc. For any length i, $|\mathcal{L}^x_i|$, is given by a 3^{rd} order linear recurrence relation:

$$\begin{aligned} &a_0 = 0, a_1 = 2, a_2 = 3 \\ &a_i = 2a_{i-1} + a_{i-2} - 2a_{i-3} \quad \text{for } i \geq 3 \end{aligned} \tag{3.5}$$

In fact, using standard techniques for solving linear homogeneous recurrences, we can derive a closed form solution to determine that

$$|\mathcal{L}^x_i| = (1/2)(2^{i+1} + (-1)^{i+1} - 1). \tag{3.6}$$

In the following discussion we give a general method based on generating functions for deriving a recurrence relation and closed form solution that we can use for model counting.

Generating Functions: Given the representation of the size of a language $\mathcal{L}$ as a sequence

$\{a_i\}$ we can encode each $|\mathcal{L}_i|$ as the coefficients of a polynomial, an ordinary generating function (GF). The *ordinary generating function* of the sequence $a_0, a_1, \ldots, a_i, \ldots$ is the infinite polynomial [61, 62]

$$g(z) = \sum_{i \geq 0} a_i z^i \tag{3.7}$$

Although $g(z)$ is an infinite polynomial, $g(z)$ can be interpreted as the Taylor series of a finite rational expression. I.e., we can also write $g(z) = p(z)/q(z)$, where $p(z)$ and $q(z)$ are finite degree polynomials. If $g(z)$ is given as a finite rational expression, each a_i can be computed from the Taylor expansion of $g(z)$:

$$a_i = \frac{g^{(i)}(0)}{i!} \tag{3.8}$$

where $g^{(i)}(z)$ is the i^{th} derivative of $g(z)$. We write $[z^i]g(z)$ for the i^{th} Taylor series coefficient of $g(z)$. Returning to our example, we can write the generating function for $|\mathcal{L}_i^x|$ both as a rational function and as an infinite Taylor series polynomial. The reader can verify the following equivalence by computing the right hand side coefficients via equation (3.8).

$$g(z) = \frac{2z - z^2}{1 - 2z - z^2 + 2z^3} = 0z^0 + 2z^1 + 3z^2 + 8z^3 + 15z^4 + \ldots \tag{3.9}$$

Generating Function for a DFA: Given a DFA A and length k we can compute the generating function $g_A(z)$ such that the k^{th} Taylor series coefficient of $g_A(z)$ is equal to $|\mathcal{L}_k(A)|$ using the transfer-matrix method [61, 62].

We first apply a transformation and add an extra state, s_{n+1}. The resulting automaton is a DFA A' with λ-transitions from each of the accepting states of A to s_{n+1} where λ is a new padding symbol that is not in the alphabet of A. Thus, $\mathcal{L}(A') = \mathcal{L}(A) \cdot \lambda$ and furthermore $|\mathcal{L}_i(A)| = |\mathcal{L}_{i+1}(A')|$. That is, the augmented DFA A' preserves both

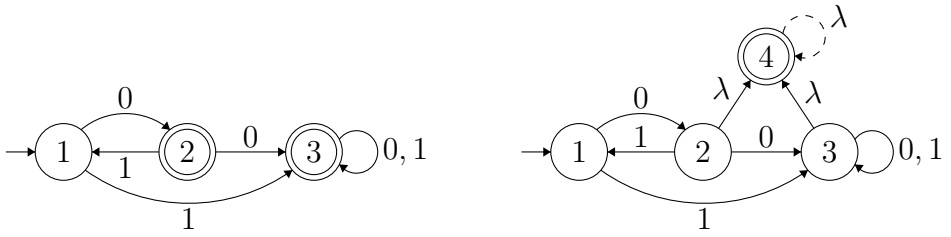


Figure 3.8: The original DFA A (left) and the augmented DFA A' (right) used for model counting (sink state omitted).

the language and count information of A. Recalling the automaton from Fig. 1, the corresponding augmented DFA is shown in Fig. 2(b). (Ignore the dashed λ transition for the time being.)

From A' we construct the $(n+1) \times (n+1)$ transfer matrix T. A' has $n+1$ states $s_1, s_2, \ldots s_{n+1}$. The matrix entry $T_{i,j}$ is the number of transitions from state s_i to state s_j. Then the generating function for A is

$$g_A(z) = (-1)^n \frac{\det(I - zT : n+1, 1)}{z \det(I - zT)}, \tag{3.10}$$

where $(M : i, j)$ denotes the matrix obtained by removing the i^{th} row and j^{th} column from M, I is the identity matrix, $\det M$ is the matrix determinant, and n is the number of states in the original DFA A. The number of accepting paths of A with length exactly k, i.e. $|\mathcal{L}_k(A)|$, is then given by $[z^k]g_A(z)$ which can be computed through symbolic differentiation via equation 3.8.

For our running example, we show the transition matrix T and the terms $(I - zT)$ and $(I - zT : n, 1)$. Here, $T_{1,2}$ is 1 because there is a single transition from state 1 to state 2, $T_{3,3}$ is 2 because there are two transitions from state 3 to itself, $T_{2,4}$ is 1 because

there is a single (λ) transition from state 2 to state 4, and so on for the remaining entries.

$$T = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, I{-}zT = \begin{bmatrix} 1 & -z & -z & 0 \\ -z & 1 & -z & -z \\ 0 & 0 & 1-2z & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}, (I{-}zT : n, 1) = \begin{bmatrix} -z & -z & 0 \\ 1 & -z & -z \\ 0 & 1-2z & -z \end{bmatrix}$$

Applying equation (3.10) results in the same GF that counts $\mathcal{L}_i(A)$ given in (3.9).

$$g_{A'}(z) = -\frac{\det(I - zT : n, 1)}{z \det(I - zT)} = \frac{2z - z^2}{1 - 2z - z^2 + 2z^3}. \tag{3.11}$$

Suppose we now want to know the number of solutions of length six. We compute the sixth Taylor series coefficient to find that $|\mathcal{L}_6^x(A)| = [z^6]g(z) = 63$.

Deriving a Recurrence Relation: We would like a way to compute $[z^i]g(z)$ that is more direct than symbolic differentiation. We describe how a linear recurrence for $[z^i]g(z)$ can be extracted from the GF. Before we describe how to accomplish this in general, I demonstrate the procedure for our example. Combining equations (3.7) and (3.11) and multiplying by the denominator, we have

$$2z - z^2 = (1 - 2z - z^2 + 2z^3) \sum_{i \geq 0} a_i z^i.$$

Expanding the sum for $0 \leq i < 3$ and collecting terms,

$$2z - z^2 = a_0 + (a_1 - 2a_0)z + (a_2 - 2a_1 - a_0)z^2 + \sum_{i \geq 3} (a_i - 2a_{i-1} - a_{i-2} + 2a_{i-3})z^i.$$

Comparing each coefficient of z^i on the left side to the coefficient of z^i on the right side,

we have the set of equations

$$\begin{aligned} a_0 &= 0 \\ a_1 - 2a_0 &= 2 \\ a_2 - 2a_1 - a_0 &= -1 \\ a_i - 2a_{i-1} - a_{i-2} + 2a_{i-3} &= 0, \quad \text{for } i \geq 3 \end{aligned}$$

One can see that this results in the same solution given in equation (3.5).

This idea is easily generalized. Recall that $g(z) = p(z)/q(z)$ for finite degree polynomials p and q. Suppose that the maximum degree of p and q is m. Then

$$g(z) = \frac{b_m z^m + \ldots + b_1 z + b_0}{c_m z^m + \ldots + c_1 z + c_0} = \sum_{i \geq 0} a_i z^i.$$

Multiplying by the denominator, expanding the sum up to m terms, and comparing coefficients we have the resulting system of equations which can be solved for $\{a_i : 0 \leq i \leq m\}$ using standard linear algebra:

$$\sum_{j=0}^{i} c_j a_{i-j} = \begin{cases} b_i, & \text{for } 0 \leq i \leq m \\ 0, & \text{for } i > m \end{cases}$$

For any DFA A, since each coefficient a_i is associated with $|\mathcal{L}_k(A)|$, the recurrence gives us an $O(kn)$ method to compute $|\mathcal{L}_k(A)|$ for any string length bound k. In addition, standard techniques for solving linear homogeneous recurrence relations can be used to derive a closed form solution for $|\mathcal{L}_i(A)|$ [69].

Counting All Solutions within a Given Bound: The above described method gives a generating function that encodes each $|\mathcal{L}_i(A)|$ *separately*. Instead, we seek a generating function that encodes the number of *all solutions within a bound*. To this end we define

the automata model counting function

$$\mathcal{MC}_A(k) = \sum_{i \geq 0}^{k} |\mathcal{L}_i(A)|. \tag{3.12}$$

In order to compute $\mathcal{MC}_A(k)$ we make a simple adjustment. All that is needed is to add a single λ-cycle (the dashed transition in Fig. 2(b)) to the accepting state of the augmenting DFA A'. Then $\mathcal{L}_{k+1}(A') = \bigcup_{i=0}^{k} \mathcal{L}_i(A) \cdot \lambda^{k-i}$ and the accepting paths of strings in $\mathcal{L}_{k+1}(A')$ are in one-to-one correspondence with the accepting paths of strings in $\bigcup_{i=0}^{k} \mathcal{L}_i(A)$. Consequently, $|\mathcal{L}_{k+1}(A')| = \sum_{i=0}^{k} |\mathcal{L}_i(A)|$ and so $\mathcal{MC}_A(k) = |\mathcal{L}_{k+1}(A')|$. Hence, we can compute $\mathcal{MC}_A$ using the recurrence for $|\mathcal{L}_i(A')|$ with the additional λ-cycle.

Case Study: Strength of Password Changing Policy

Consider a scenario in which a system user's password has been compromised and they must select a new password. However, the password changing policy will reject the new password if it is too similar to the old password. This similarity comparison is performed with the function in Figure 3.9. Furthermore, suppose an attacker knows the old password, knows, that the user must change their password, and also knows the new password policy. With this information, an attacker can rule out many possibilities for the user's updated password. How much can an attacker rule out? I demonstrate how automata-based model counting can answer these types of questions.

Suppose an adversary knows `old_p = "abc-16"`. By performing symbolic execution, we can extract constraints on the value of the new password, as shown in Figure 3.10, with symbolic variable `NEW_P`. Using automata-based solving we construct a DFA that

```
1  public Boolean NewPWCheck(String new_p, String old_p){
2     if( old_p.contains(new_p)             || ...
3         new_p.contains(old_p)             || ...
4         old_p.reverse().contains(new_p))  || ...
5         new_p.contains(old_p.reverse())     ){
6         System.out.println("Too similar.");
7         return false;
8     } else
9         return true;
10 }
```

Figure 3.9: Code for a password changing policy. If the new password is too similar to the previous password it, the change is rejected.

```
(not (contains (toLower NEW_P) "abc-16"))
(not (contains (toLower NEW_P) "61-cba"))
(not (contains "abc-16" (toLower NEWP)))
(not (contains "61-cba" (toLower NEWP)))
```

Figure 3.10: Constraints on new password given the old password ("abc-16") and the password updating policy.

characterizes all solutions for `NEW_P` which has 36 states (not counting the sink state)—too large to illustrate in this dissertation. The transfer matrix for the DFA is shown in Figure 3.11. Although the DFA has 36 states, the transition matrix is very sparse. We observe that this is often the case in practice. Given the transition matrix, we can then compute the generating function which enumerates `NEW_P`, and the series expansion of the first few terms:

$$g(z) = \frac{8096z^{12} - 8128z^{11} + 32z^{10} + 16z^7 - 16z^6 - 256z^2 + 257z - 1}{194304z^{17} + 225920z^{16} + 241984z^{15} + \ldots + z^5 - 6114z^4 - 2280z^3 - 247z^2}$$

$$g(z) = 247z^2 + 65759z^3 + 16842945z^4 + 4311810213z^5 + 1103823437965z^6 + \ldots$$

If the length of the password is n, then there are $|\Sigma|^n$ possible passwords, but knowing the policy, the search space is reduced to the number of models for the extracted constraints, given by the n-th series coefficient of $g(z)$. For instance, for passwords of length 6 brute force searching all possible passwords has a search space of $256^6 = 2^{48}$. If adversary knows `old_p` and the policy, then there are $[z^6]g(z) = 1103823437965 \approx 2^{40.0056}$ passwords. Thus, the search space is reduced by about a factor of $2^{7.9944}$.

3.2.4 Implementation of Automata-Based Model Counting

We implemented Automata-Based model Counter for string constraints (ABC) using the symbolic string analysis library provided by the Stranger tool [70, 71, 72]. We used the symbolic DFA representation of the MONA DFA library [73] to implement the constructions given in Algorithm 5. In MONA's DFA library, the transition relation of the DFA is represented as a Multi-terminal Binary Decision Diagram (MBDD) which results in a compact representation of the transition relation.

ABC supports the SMT-LIB 2 language syntax. We implemented automata-based model counting by passing the automaton transfer matrix to Mathematica for computing the generating function, corresponding recurrence relation, and the model count for a specific bound. Because the DFAs we encountered in our experiments typically have sparse transition graphs, we make use of Mathematica's powerful and efficient implementations of symbolic sparse matrix determinant functions [74].

3.2.5 Comparison with Syntax-Based Model Counting

SMC Examples. For a comparative evaluation of our tool with SMC, we used the examples that are listed on SMC's web page. We translated the 6 example constraints listed in table 3.3 into SMT-LIB2 language format that we support. We compare our

$$\begin{pmatrix}
0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{2} & \mathbf{2} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1}
\end{pmatrix}$$

Figure 3.11: Transfer counting matrix for DFA for all possible values of `NEW_P`. Notice the sparsity.

results with the results reported in SMC's web page. The first column of the Table 3.3 shows the file names of these example constraints. The second column shows the string length bounds used for obtaining the model counts. The next two columns show the log-scaled SMC lower and upper bound values for the model counts. The last column shows the log-scaled model count produced by ABC . We omit the decimal places of the numbers to fit them on the page. For all the cases ABC generates a precise count given the bound. ABC's count is exactly equal to SMC's upper bound for four of the examples and is exactly equal to SMC's lower bound for one example. For the last example ABC reports a count that is between the lower and upper bound produced by SMC. Note that these are log-scaled values and actual differences between a lower and an upper-bound values are huge. Although SMC is unable to produce an exact answer for any of these examples, ABC produces an exact count for each of them.

Table 3.3: Log-scaled comparison between SMC and ABC.

	bound	SMC lower bound	SMC upper bound	ABC count
nullhttpd	500	3752	3760	3760
ghttpd	620	4880	4896	4896
csplit	629	4852	4921	4921
grep	629	4676	4763	4763
wc	629	4281	4284	4281
obscure	6	0	3	2

3.2.6 Automata-Based Counting for Linear Integer Constraints

In order to handle LIA constraints in ABC, we implemented automata construction techniques for linear arithmetic constraints on integers [75]. The approach we use can handle arithmetic constraints that consist of linear equalities and inequalities ($=, \neq, >, \geq, \leq, <$) and logical operations ($\wedge, \vee, \neg$). As these constraints can always be written in the form $Ax \leq b$, our method handles the same language of constraints that LattE and

Barvinok can handle (although we do not support symbolic model counting in the same way that Barvinok does.)

Similar to string-constraint solving, the goal is to create an automaton that accepts solutions to the given formula. However, for numeric constraints, it is necessary to keep relationships between multiple integer variables in order to preserve precision. For example, given a numeric constraint such as $2x - y = 0$, we would like the automaton to recognize the tuples (x, y) such that $(x, y) \in \{(0, 0), (1, 2), (2, 4), (3, 6), \ldots\}$. If we separate the set of values for x and y and recognize the set $0, 1, 2, 3, \ldots$ for x and the set $0, 2, 4, 6, \ldots$ for y, then we would get tuples such as $(2, 2)$, which are not allowed by the constraint $2x - y = 0$. To address this, we use multi-track automata which are a generalization of finite state automata. A multi-track automaton accepts tuples of values by reading one symbol from each track in each transition. I.e., given an alphabet Σ, a k-track automaton has an alphabet Σ^k.

For numeric constraints, we use the alphabet $\Sigma = \{0, 1\}$. The numeric automata accept tuples of integer values in binary form, starting from the least significant digit.

We implement an automata constructor function $\mathcal{A}$ for numeric constraints, such that, given a numeric constraint F, $\mathcal{A}(F)$ is an automaton where $\mathcal{L}(\mathcal{A}(F)) = [\![F]\!]$. Note that, for numeric constraints, $\mathcal{A}(F)$ accepts tuples of integer values, one for each variable in the constraint F. Each variable in F is mapped to a unique track of the multi-track automaton that we construct.

The automata constructor $\mathcal{A}$ for numeric constraints handles the boolean operators $\neg, \wedge, \vee$ the same way as the automata constructor for string constraints. Each basic numeric constraint is in the form $\sum_{i=1}^{n} a_i \cdot x_i + a_0 \; op \; 0$, where $op \in \{=, \neq, >, \geq, \leq, <\}$, a_i denote integer coefficients and x_i denote integer variables. The automata construction for basic numeric constraints relies on a basic binary adder state machine construction [75]. The state machine starts from a state labeled with the constant term a_0. It reads the first

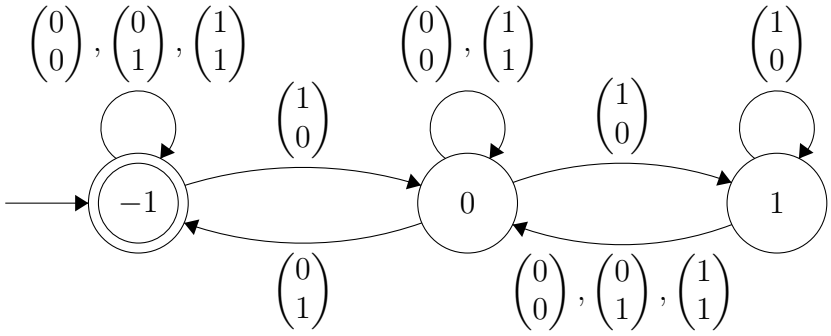


Figure 3.12: Automata for the numeric constraint $x - y < 1$.

binary digit of all the variables, computes the result of the sum for the first digit and the carry. The next state is the state that corresponds to the new carry. Using each digit and the current carry, it is possible to compute the next carry which define the transitions of the state machine. Accepting states are determined based on the operation *op*. For example, if the operation is $=$, then all the resulting digits should be equal to 0 and the carry should also be 0. So the state 0 is accepting and all transitions that result in a non-zero digit go to the sink state. In order to handle negative values, 2's-complement representation is used.

As an example, in Figure 3.12 we show the multi-track automaton that accepts tuples of integer values that satisfy the constraint $x - y < 1$ (the transitions are labeled with the digit for variable x on top of the digit for variable y).

Once a numeric DFA is constructed for an LIA constraint, model counting is accomplished in the same way as for strings, but with a slightly different interpretation. Counting the number of accepting paths of length k in the DFA corresponds to counting the number of solutions to the constraint that use k bits to represent integers. Thus, our counting method prodces a generating function such that $[z^k]g(z)$ is the number of solutions to C that contain at most k bits. For the example DFA in Figure 3.12, the transfer matrix and generating function are given below.

$$T = \begin{pmatrix} 3 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 3 & 1 \end{pmatrix} \qquad g(z) = \frac{3}{3z^2 - 4z + 1}$$

3.3 Chapter Summary

In this chapter I discussed previous work in model counting for Boolean, integer, and string constraints using a variety of methods. Then I discussed automata-based methods for model counting string and integer constraints using linear algebra, graph theory, and generating functions. In the next chapters, we will see how model counting can be used along with symbolic execution and information theory in order to quantify side-channel vulnerabilities and synthesize optimal side-channel attacks.

Chapter 4

Side-Channel Analysis for Segmented Oracles

This chapter addresses functions which behave as *segmented oracles*. Segmented oracle side channels occur when a side-channel observation allows an attacker to infer information about segments of secret data, like a string prefix or an array slice. The segments can be adaptively explored and information about each segment combined which results in efficient side-channel attacks. Time-based segmented oracles can result from library functions that use early termination optimizations for string, array, and memory equality comparisons, which are present in many programming languages including C, C++, Java, Python, Ruby, and PHP [18, 28, 76, 17]. As described in [77], these types of library functions have enabled real-world segmented oracle attacks against the Xbox 360 operating system [5], and the hash-message authentication code (HMAC) comparisons in the Google Keyczar cryptographic library [6] and the open authorization standards OAuth [78, 79] and OpenID [80]. I demonstrate the applicability of the approach given in Section 4.3 on a password verification function as an example of an early termination segmented oracle and this analysis applies equally well to the other examples just de-

scribed. On the other hand, *size-based* segmented oracles can arise from text compression functions [19] resulting in leakage of confidential web-session information by measuring the sizes of files and network communications [20]. I demonstrate this approach for this type of segmented oracle using LZ77 compression [1]. Although the approach I present in Section 4.2 requires an ordered traversal of the secret's segments (which is the case for all the examples listed above), I believe that, in the future, it can be generalized to handle oracles which do not require a specific ordering of segments.

The research contributions of this chapter can be summarized as follows: 1) Single-run side-channel analysis using SPF that computes the information leakage in terms of Shannon entropy using probabilistic symbolic execution and listeners that track the observable values such as execution time, file size, or memory usage. 2) Two types of multi-run side-channel analysis for segmented oracles based on a best-adversary model. The first approach composes the adversary model and the function under analysis within a loop and conducts the multi-run analysis on the composed system. However, this approach leads to path explosion. I also present a second, more efficient, approach for multi-run side-channel analysis for segmented oracles that uses path constraints generated for only a single-run symbolic execution of the function. 3) I extend SPF to enable analysis of Java programs that manipulate strings using two approaches. One of them traces the implementations of string manipulation functions and treats strings as bounded arrays of characters that are represented as bit-vectors, and checks satisfiability of path constraints using the SMT solver Z3. The second approach generates constraints in the theory of strings and uses the string constraint solver ABC, described in Chapter 3. 4) I use two model counting constraint solvers for computing information leakage. One of them is LattE, which has been used for analyzing numeric constraints in SPF before. In this chapter, I desctibe our extension of the SPF+LattE framework to the analysis of string constraints by viewing strings as arrays of characters. I also integrate the

```
1 public F1 (char[] h, char[] l){
2   for (int i = 0; i < h.length; i++)
3     if (h[i] != l[i]) return false;
4   return true;
5 }
```

Figure 4.1: Password-checking function F_1.

```
1 public F2 (char[] h, char[] l){
2   matched = true;
3   for (int i = 0; i < h.length; i++) {
4     if (h[i] != l[i]) matched = false;
5     else matched = matched;
6   }
7   return matched;
8 }
```

Figure 4.2: Password-checking function F_2.

model-counting string constraint solver ABC with SPF. 5) I conduct experiments on two side-channel examples, demonstrating the performance of different approaches.

4.1 Segment Oracles

Consider a password-based authentication function. The password checking function has two inputs: 1) a password, which is secret, and 2) a user input, which is public. The function should compare the password and the user input and return true if the input matches the password and return false otherwise; it should not leak any information about the password if the input does not match.

Let us consider two password checking functions F_1 and F_2 whose implementations are shown in Figures 4.1 and 4.2, respectively. I assume that functions F_1 and F_2 are executed on inputs h and l, where I follow the typical notation used in the security literature: h denotes the *high* value, i.e. the secret password, and l denotes the *low* value, i.e. the public input that the function compares with the password.

Both functions return true or false indicating if the input (l) matches the secret (h).

Note however that the functions may leak some information about the secret through *side channels*, in this case an adversary may infer some information about the secret h by measuring the execution time (as explained below). In general, let us assume that function $F(h, l)$ returns an *observable* value o which represents the side-channel measurements of the adversary after executing $F(h, l)$. The observable value o can be one of a set of observable values O. Let us assume that the observable values are noiseless, i.e., multiple executions of the program with the same input value will result in the same observable value.

For the functions F_1 and F_2 above, let us use the execution time as an observable. For function F_1 this will result in $n + 1$ observable values where n is equal to the length of h, i.e., $o \in \{o_0, o_1, \ldots o_n\}$, since function F_1 will have a different execution time based on the length of the common prefix of h and l. If h and l have no common prefix, then F_1 will have the shortest execution time (let us call this observable value o_0) since the loop body will be executed only once (assuming the password length is not zero). If h and l have a common prefix of one character (and assuming that password length is greater than or equal to two), then F_1 will have a longer execution time since the loop body will be executed twice (let us call this observable value o_1). In fact, for each different length of the common prefix of h and l, the execution time of F_1 will be different. Let observable value o_i denote the execution time of F_1 for the common prefix of size i. Note that o_n corresponds to the case where h and l completely match where n is the length of the password. On the other hand, execution time of F_2 is always the same, so F_2 does not have a side channel.

Side-channel analysis can be used to answer following type of questions: Are F_1 and F_2 leaking information about the secret through side channels, and, if so, how much? Based on the above discussion, we can see that F_1 is leaking information about h even when h and l do not completely match. By observing the execution time of F_1, an

adversary can deduce the length of the common prefix of h and l.

F_2 on the other hand leaks no information through the side channel. Note that an attacker learns that h is not equal to the value of l. However, the information leakage for F_2 is pretty small compared to the information leakage by F_1 (which has the timing side channel). For example, assuming the secret is a four digit PIN, an attacker needs at most 10^4 tries to guess the password using F_2 but at most 40 tries using F_1 (as the adversary can try to first guess the first digit in the PIN, then secod digit etc. using the side-channel information). The question is: can we formalize the amount of information leaked and can we automatically compute it?

4.2 Entropy Computation

I introduced Shannon entropy in Chapter 2 and now describe how it applies to the current problem at hand. The observables produced by a function F can be considered the messages that an adversary receives by executing F; they correspond to messages about the secret. Hence, we can use the Shannon entropy to measure the amount of information conveyed by each execution of F, i.e., the amount of information leaked by function F. Define the Shannon entropy of a function F as:

$$\mathcal{H}(F) = -\sum_{o_i \in O} p(o_i) \times \log_2(p(o_i))$$

where $p(o_i)$ is the probability of observing the value o_i after executing F. In order to compute the Shannon entropy, we need to compute the probability of observing each value o_i. We can compute these probabilities using probabilistic symbolic execution with model counting [81], (see Chapter 2.

In this analysis I perform symbolic execution (where both h and l are symbolic) to

systematically analyze all paths through the code. In this way, we collect each path condition and corresponding observable as an ordered pair (o_i, PC_i). For example, symbolic execution of function F_1 results in a set of $n+1$ path conditions, each with a different time observation: $\{(o_0, PC_0), (o_1, PC_1), \ldots (o_n, PC_n)\}$, with $o_0 < o_1 < \ldots < o_{n+1}$, and path conditions of the form:

$$\begin{array}{lcl} PC_0 & \equiv & h[0] \neq l[0] \\ PC_1 & \equiv & h[0] = l[0] \wedge h[1] \neq l[1] \\ PC_2 & \equiv & h[0] = l[0] \wedge h[1] = l[1] \wedge h[2] \neq l[2] \\ & \vdots & \\ PC_{n-1} & \equiv & h[0] = l[0] \wedge h[1] = l[1] \wedge \ldots \wedge h[n] \neq l[n] \\ PC_n & \equiv & h[0] = l[0] \wedge h[1] = l[1] \wedge \ldots \wedge h[n] = l[n] \end{array}$$

Each path condition PC_i encodes the fact that a prefix of the public input l matches a prefix of the secret h. For example (o_1, PC_1) indicates that the first character in the public input matches the first charter in the secret and the second character does not match. For the largest observable, we see that (o_n, PC_n) indicates that the public and private inputs match on all segments.

One can compute the probability $p(o_i)$ for each observable value o_i, using model counting over the path conditions, in the following way. Let D denote the input domain (i.e., the set of possible values for h and l, assumed to be finite) and let $|D|$ denote the size of the input domain. Let us write $|PC|$ to denote the number of solutions over D that satisfy the path constraint PC. One can compute $|PC|$ using a model counting constraint solver [24, 25]. Assuming a uniform distribution for h and l the probability of observing o_i is $p(o_i) = |PC_{o_i}|/|D|$ where the probability of the input value completely matching the password h is $p(o_n)$.

For function F_2 there are only two observable values through the *main channel*, i.e. the boolean values returned by the function, and the corresponding path constraints are:

- $\neg(h[0] = l[0] \wedge h[1] = l[1] \wedge .. \wedge h[n-1] = l[n-1])$
- $h[0] = l[0] \wedge h[1] = l[1] \wedge .. \wedge h[n-1] = l[n-1]$

Figure 4.3 shows the Shannon entropy computed for F_1 and F_2 as described above using probabilistic symbolic execution and model counting. Note that as the size of the password increases, the entropy gets very close to 0 for function F_2. So, for a reasonable sized password, F_2 does not leak information. However, for F_1 we observe that the information leaked remains slightly above 1 bit even if we keep increasing the length of the password. Independent of the size of the password, F_1 leaks information about the first digit of the password due to the timing side channel.

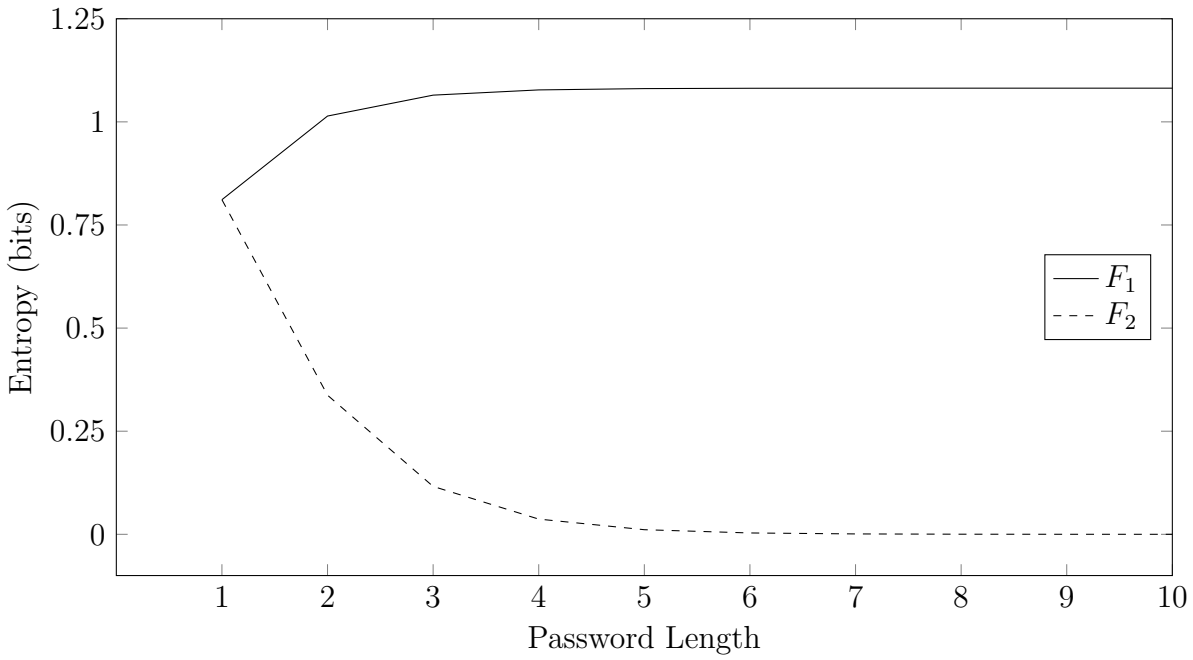


Figure 4.3: Entropy after a single guess for functions F_1 and F_2, for password length ranging from 1 to 10.

The analysis I presented above computes the amount of information leaked by a single execution of a function. One can also easily determine the amount of initial information

in the system by assuming that h is picked using a uniform distribution from the domain D_h. Then the amount of information in the system initially is:

$$\mathcal{H}(h) = -\sum_{v \in D_h} 1/|D_h| \times \log_2(1/|D_h|) = \log_2(|D_h|)$$

An execution of the function leaks the amount of information given by the Shannon entropy of the function $\mathcal{H}(F)$ and the remaining entropy in the system is $\mathcal{H}(h) - \mathcal{H}(F)$.

An interesting question to answer is the following: How many tries would it take an adversary to figure out the password? We can try to estimate the attack sequence length using the information leakage. When the amount of information in the system reaches zero, then, we can conclude that the adversary has figured out the password.

Based on the amount of initial information and the Shannon entropy for the function, one can try to estimate the amount of runs it would take an adversary to determine the secret. However, this analysis would not be accurate since an adversary could learn from previous tries and choose the l values accordingly based on earlier observations. So, except for the first run, the adversary would not pick the l values with a uniform distribution from the domain of l. In order to do a more precise analysis one needs to model the adversary behavior. I discuss how to do this for a particular class of problems called segmented oracles in the following section.

4.3 Multi-run Analysis of Segment Oracle Attacks

Segmented oracle side channels provide observations about "segments" of the secret. For example, a segmented oracle side channel can provide an observable value (such as execution time) that enables an adversary to determine if the first character of the secret value (for example a password) matches to the public value (the input provided by the

adversary). In general, a segmented oracle provides a distinct value for each matched segment (such as the matching the first character in the password, matching the first 2 characters, the first 3 characters, etc.) I relaxe this assumption in Chapter 6.

Note that for the function F_1 shown in Figure 4.1, execution time serves as a segmented oracle side channel. The function terminates the execution immediately if it determines that the first character of the secret does not match the input and the execution time increases linearly with the number of segments that match. I.e., by observing the execution time, an adversary can figure out how many characters of the secret match the public input. Hence, for the function F_1, execution time acts as a segmented oracle side channel.

The function F_1 is a particular instance of an early termination optimized equality comparison. It returns false *as soon as* it discovers a mismatch in order to avoid unnecessary comparisons. This is a common programming pattern found in many library functions which results in segmented oracle timing attacks [18, 28, 76, 17, 77]. These vulnerabilities are remedied by implementing constant time functionally equivalent versions of those comparison functions that operate over sensitive data, for example F_2, in order to remove the timing side channel [76, 6, 28, 17]. The approach in this chapter provides a method for automatically quantifying the amount of information an advesary can gain by a function under a segmented oracle side channel attack, indicating whether a constant time implementation is necessary.

Now, let us discuss the adversary model. Let us assume that the adversary runs a function F multiple times with different l values (but the secret h stays the same) and records the corresponding observables, while trying to figure out h. Further let us assume that the analyzed programs are deterministic, i.e. given h and l values, $F(h, l)$ returns one observable value o which represents the observations of the adversary after executing $F(h, l)$. For segmented oracles the observable values consist of a set of values

Algorithm 6 Adversary-Function Systems, $S(A, F)$

```
1: procedure S(A, F)
2:     seq ← nil
3:     repeat
4:         l ← A(seq)
5:         o ← F(h, l)
6:         seq ← APPEND(seq, ⟨l, o⟩)
7:     until (o = o_n)
```

$o \in \{o_0, o_1, \ldots o_n\}$ where o_0 denotes no segments of the input (l) and secret (h) match, o_i denotes i segments of the secret match the input, and o_n denotes the secret completely matches the input.

Let us call each execution of F a run. So, the adversary is generating a sequence of l values to run the program multiple times, the intuition being that each run reveals some new information about the secret. We can formalize the adversary as a function A that takes all the prior history as input (which is a sequence of tuples where each tuple is a l value and the corresponding observable for the execution of function F with that l value). Note that the h value is constant and does not change from one execution to the other.

One can model the whole system $S = (A, F)$, where the adversary A generates l values for multiple executions of the function F in order to determine the secret h, as in Algorithm 6. Given the system $S = (A, F)$ one may want to compute the probability of determining the secret after k runs, i.e., having $|seq| = k$ when S terminates. Or, one may want to compute the information leakage (i.e., entropy) for k runs. One approach would be to analyze the system S without restricting the adversary. However, this would take into account behaviors such as the adversary trying the same l value over and over again even though it does not match the secret. When analyzing vulnerabilities of a software system, we have to focus on the behavior of the best adversary.

For the segmented oracles, it is easy to specify the best adversary A_B [3]. This adver-

sary works as follows: Let $\langle l^1, o^1\rangle, \langle l^2, o^2\rangle, \ldots, \langle l^k, o^k\rangle$ be the run history. The adversary generates l^{k+1} for the $k+1$st run as follows:

- If $o^k \neq o^{k-1}$ and $o^k = o_i$, then the adversary constructs l^{k+1} as follows: $\forall j, 1 \leq j < i \ : \ l^{k+1}[j] = l^k[j]$ (part of l that already matched remains the same), $l^{k+1}[i] \neq l^k[i]$, (use a different value for the first part that did not match in the last try) and rest of the l^{k+1} is randomly generated.

- If $o^k = o^{k-1}$, then let m be the smallest number where $o^m = o^k$ and let $o^k = o^m = o_i$, then the adversary constructs the l^{k+1} as follows: $\forall j, 1 \leq j < i \ : \ l^{k+1}[j] = l^k[j]$ (part of l that already matched remains the same) and $\forall j, m \leq j < k \ : \ l^{k+1}[i] \neq l^j[i]$ (use a different value then the values that have already been tried for the first part that does not match) and rest of the l^{k+1} is randomly generated.

Let S^k denote the execution of the system $S = (A_B, F)$ where the function F is executed k times, i.e., $|seq| = k$. One can ask the following question: What is the probability of the adversary A_B guessing the password in exactly k tries?

Note that, execution of S^k will generate observable sequences $o^1, o^2, \ldots, o^k$ where for all $1 \leq t \leq k, o^t = o_i \wedge o^{t+1} = o_j \Rightarrow j \geq i$. I.e., since we are using the best adversary model A_B, the observable values in the sequence will be non-decreasing. The adversary will never produce a worse match than the one in the previous try. Another constraint for the observable sequences is that if o_n appears in a sequence, then o_n is the last observable of the sequence since S terminates when o_n is observed.

One can calculate the probability of determining the password in exactly k tries as the probability of generating the observable sequences $o^1, o^2, \ldots, o^l$ where $l \leq k$, observable values in the sequence are non-decreasing, and $o^l = o_n$.

Let $p(o^1, o^2, \ldots, o^k)$ denote the probability of S^k generating that particular observable sequence. Then one can compute the entropy for S^k (i.e., the information leakage within

the first k runs) as follows:

$$\mathcal{H}(S^k) = -\sum_{o^1,o^2,\ldots,o^l \in SEQ^k} p(o^1, o^2, \ldots, o^l) \times \log_2(p(o^1, o^2, \ldots, o^l))$$

where SEQ^k is the set of all non-decreasing observable sequences that can be generated by the first k iterations of $S = (A_B, F)$. For every sequence $o^1, o^2, \ldots, o^l \in SEQ^k$: 1) $l \leq k$, 2) the observable values in the sequence are non-decreasing, 3) if o_n appears in the sequence, then it is the last observable in the sequence, and 4) if o_n does not appear in the sequence, then $l = k$.

Here I present two approaches to multi-run analysis of segmented oracles. The first approach is intuitive and more general; it is applicable to any adversary model. However, this approach requires the probabilistic symbolic execution of an adversary model which executes the program multiple times, and thus it suffers from the path explosion problem. To address this problem, for the best adversary model, I propose a more scalable approach with a novel computation of the leakage which requires the probabilistic symbolic execution on only one run of the program.

4.3.1 Multi-Run Symbolic Execution

The first approach for multi-run side-channel analysis is described with the following two steps. First, let us create a model of the attack scenario, explained in Section 4.1, where an adversary can provide the low inputs, and execute the program a number of times. Then, one can use probabilistic symbolic execution to explore all possible observations of the model and compute the probability for each observation. Shannon entropy and channel capacity of the leaks are easily derived from the probabilities.

In this work, a model in the analysis is a Java bytecode program, written as a driver for the program under test. Since the secret h and the inputs of the adversary $l^1, \ldots l^k$ are

not known in advance, they are modeled by symbolic variables in symbolic execution. Without any constraints on $l^1, \ldots l^k$, this is a model for a very naive adversary, who repeatedly tries to guess the secret with random values, and learns nothing from the previous attempts.

To model an adversary who gains information through observing program executions and revises the input domain accordingly, let us use the assume-guarantee reasoning in symbolic execution to impose the constraints on the inputs. Let us illustrate the approach by implementing a particular adversary model.

4.3.2 The Best Adversary Model

Algorithm 7 depicts a driver S modeling the best adversary described in section 4.1. Here an observation o^i of the adversary indicates how many segments in the low input matched with the secret. The adversary is allowed to make k executions of $F(h, l)$ but stops early if all the segments are matched, i.e. $o^i = |h|$. The instruction ***assume***, implemented by the built-in API `Debug.assume` in SPF, is used to impose constraints on the inputs.

The best adversary is characterized by two sets of assumptions. The first set of assumptions reflect the fact that, for the segment being search s, the best adversary selects an input different from the ones in the previous executions. When the adversary discovers more segments of the secret, i.e. when $o^i > o^{i-1}$, they keep these constant segments for the inputs of the following executions, and move on to search for the next segment. This is modeled by the second set of assumptions in the procedure.

Algorithm 7 $S = (A_B, F)$, Composed System of Best Adversary and Function

1: s: the current segment of h being searched
2: b: the first time s is searched
3: $o^0, o^1, \ldots o^k$: observations of the adversary
4: $s \leftarrow 1$, $b \leftarrow 1$, $o^0 \leftarrow 0$
5: **for all** $i \in [1..k]$ **do**
6: **for all** $j \in [b..i)$ **do** ***assume*** $(l^i[s] \neq l^j[s])$
7: $o^i \leftarrow F(h, l^i)$
8: **if** $(o^i = |h|)$ **then return**
9: **if** $(o^i > o^{i-1})$ **then**
10: **for all** $j \in [i+1..k]$ **do**
11: **for all** $n \in [s..o^i]$ **do** ***assume*** $(l^j[n] = l^i[n])$
12: $s \leftarrow o^i + 1$, $b \leftarrow i + 1$

4.3.3 Computation of Information Leakage

In this approach, the computation of leakage does not depend on any particular adversary model $S = (A, F)$, i.e. it can be applied to any model with any assumptions made by the adversary, or even no assumptions at all.

For this analysis, we extend classical symbolic execution to keep track of the assumptions ASM in a symbolic path. At a low level, ASM is implemented with exactly the same data structure as the path condition. When executing the instruction ***assume***(c), symbolic execution updates the path condition $PC \leftarrow PC \wedge c$, and checks satisfiability with a constraint solver. Symbolic execution advances to the next instruction if the updated PC is satisfiable, and it backtracks otherwise. This extension for symbolic execution updates $ASM \leftarrow ASM \wedge c$ only when the updated PC is satisfiable. Thus, there is no constraint solving overhead for ASM.

SPF performs symbolic execution, with the extension, on the model $S = (A, F)$ to explore all possible observations. Each observation of S is a sequence of observations of F: $\overrightarrow{o_i} = \langle o^1, o^2 \ldots o^n \rangle$ where $1 \leq n \leq k$. For each $\overrightarrow{o_i}$, we also obtain from symbolic execution the path condition PC_i that leads to that observation, and the assumptions

ASM_i on that path.

Let us denote by $D_h, D_1, D_2 \ldots D_k$ the domains of $h, l_1, l_2 \ldots l_k$ respectively. The input space is then $D = D_h \times D_1 \times \cdots \times D_k$. If there is no assumptions on the low inputs, l_i can take any value D_i. Hence, the search space of the adversary is D, and the probability of observing $\overrightarrow{o_i}$ is computed by

$$p(\overrightarrow{o_i}) = \frac{|PC_i|}{|D|}$$

In the case the adversary has some knowledge about the input, modeled by the assumptions, the revised domain of $\overrightarrow{o_i}$ is $|ASM_i|$, and hence its probability is

$$p(\overrightarrow{o_i}) = \frac{|PC_i|}{|ASM_i|}$$

Both $|PC_i|$ and $|ASM_i|$ are computed by model counting tools integrated in probabilistic symbolic execution as discussed in the following sections.

4.4 Multi-Run Analysis Using Single-Run Symbolic Execution

As shown in the previous section, one is able to compute the probabilities of observation sequences by performing a complete symbolic execution of a program which simulates the adversary strategy of repeated guessing. However, performing a complete symbolic execution over all iterations of adversary behavior can become prohibitively expensive. Therefore, I seek to avoid this expensive computation. In this section, I describe how to compute the sequence probabilities using symbolic execution and model counting from only a single iteration of the adversary strategy, by taking advantage of the segmented nature of observations which reveal the secret.

Notation. For a segmented oracle the low (l) and high (h) inputs are compared incre-

mentally. The n segments of l and h are denoted by $l[0], \ldots, l[n-1]$ and $h[0], \ldots, h[n-1]$, respectively. We write $h[i:j]$ for the "slice" of h from index i to index j, and similarly for l. Let D_i be the domain size of $l[i]$, or equivalently, the domain size of $h[i]$, and write $\mathbf{D} = \langle D_0, D_1, \ldots, D_{n-1} \rangle$ for the vector of these domain sizes. Le us write $\mathbf{D}_{i:j}$ to denote the subvector of $\mathbf{D}$ of indices i through j, and $\prod \mathbf{D}$ for the product of all elements of $\mathbf{D}$.

Probability Computation. By performing a symbolic execution of a single run of $F(h, l)$ one can automatically generate the set of observables and corresponding path conditions, $\{(o_i, PC_i) : 0 \leq i \leq n\}$. Without loss of generality assume an order of observables, $o_0 < o_1 < \ldots < o_{n+1}$, and assume that the path conditions are in the form given below, a generalization of the path conditions given in Section 2. Path constraints of this form result from symbolic execution of comparison functions which utilize the early termination optimization programming pattern, as described in Section 2.

$$PC_i \equiv \begin{cases} (l[i] \neq h[i]) \wedge \left(\bigwedge_{j=0}^{i-1} h[j] = l[j] \right) & \text{if } i < n \\ \bigwedge_{j=0}^{n-1} h[j] = l[j] & \text{if } i = n \end{cases}$$

Due to the segmented nature of the comparison between l and h, one can consider the size of the domain D_i for each segment, that is, the number of possible values to which each segment can be assigned, independently. Then each PC_i determines a combinatorial restriction on the set of $\mathbf{D}$.

- In the case of PC_n where each $h[i] = l[i]$, for any of the D_i values for $l[i]$, the value of $h[i]$ is constrained to a single value. Therefore, the product of the domain sizes must be equal to $|PC_n|$.
- For PC_i $(i < n)$, $h[j] = l[j]$ for $j < i$, and so for any of the D_i values for $l[j]$, $h[j]$ is constrained to be a single value. Since, $h[i] \neq l[i]$, for any of the D_i values for $l[i]$, there are $D_i - 1$ possible values for $h[i]$. Finally for $j > i$ there is no constraint on

the relationship between $l[i]$ and $h[i]$ and so there are D_i possible values for each of them.

The combinatorial argument above can be summarized by the following system of equations:

$$\begin{cases} \prod \mathbf{D} = |PC_n| \\ \prod \mathbf{D} \cdot (w_i - 1) \cdot \prod \mathbf{D}_{i+1:n-1} = |PC_i| \end{cases}$$

This system of equations can be solved for each w_i via reverse substitution using the following recurrence:

$$D_i = \frac{|PC_i|}{|PC_n| \cdot \prod \mathbf{D}_{i+1:n-1}} + 1$$

Once we have determined the domain sizes of the individual segments, we are in a position to compute the probability any particular observation sequence. Let $p(\overrightarrow{o}|\mathbf{D})$ be the probability of observation sequence $\overrightarrow{o}$ given a vector of segment domains $\mathbf{D}$. In addition, define $\mathbf{D}'_i$ to be the vector of domains constrained by PC_i. That is $\mathbf{D}'_i = \langle 1, 1, \ldots, D_i - 1, D_i + 1, \ldots, D_n \rangle$. Then $p(\overrightarrow{o}|\mathbf{D})$ can be computed recursively using the following logic:

- Base Case: if $\overrightarrow{o} = o_i$ is a sequence of length 1, the probability of o_i is $(\prod \mathbf{D}'_i) \,/\, (\prod \mathbf{D})$, that is, the number of remaining possible inputs that are consistent with (o_i, PC_i), out of the total number of inputs in the domain.
- Recursive Case: if $\overrightarrow{o} = \langle o^1, o^2, \ldots o^k \rangle$ is a sequence of length k one can think of it as o^1 followed by a sequence of length $k - 1$. Then computing $p(\overrightarrow{o}|\mathbf{D})$ reduces to computing the probabilities of $p(o^1|\mathbf{D}'_i)$ and $p(\langle o^2, \ldots, o^k \rangle|\mathbf{D}'_i)$ and multiplying.

The above presented computation results in the same probabilities that are computed by a full probabilistic symbolic execution analysis of the adversary's complete attack

behavior. Given the probabilities, one can simply apply the entropy formula. Both methods have been implemented and it was experimentally verified that they produce the same results. However, the second method is significantly faster. I discuss this in Section 4.5 containing the experimental results.

4.5 Experiments

To validate the effectiveness of these methods, we first evaluated ABC by comparing it with LattE. Next we compare the efficiency of using multi-run vs. single-run symbolic execution for computing the entropy after a sequence of observations. Lastly, we have tested this side-channel analysis on: 1) the password checking function described in Section 3 which is susceptible to a timing attack, and 2) a compression function which contains a side channel based on the size of the compressed output file.

4.5.1 Timing Performance of Model Counting

Symbolic PathFinder already contained an implementation of path constraint model counting using LattE [32]. In addition, we integrated ABC in SPF for counting solutions to path constraints. The experiments show that ABC and LattE produce identical model counting results. To compare running time, we analyzed the password checking function. We compare the end-to-end running time of performing symbolic execution, collecting path constraints, and performing model counting on all generated constraints in order to compute the information leakage of a single run by the adversary. We find that the implementation using ABC is significantly faster than the previous implementation that uses Latte. As shown in Figure 4.4, for a fixed alphabet size of 4, we see that the running time increases with the password length for both ABC and LattE, and that the ABC

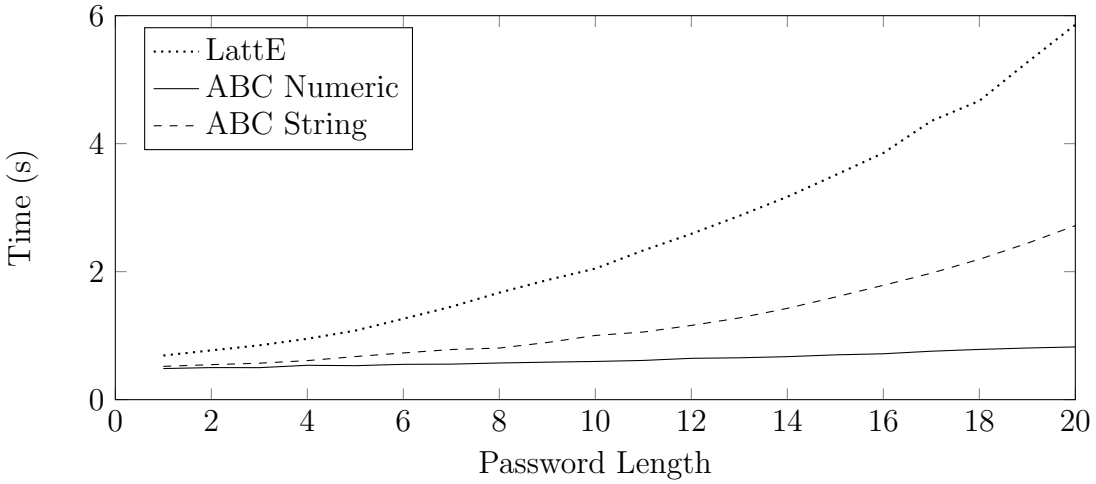


Figure 4.4: Time comparison for computing single guess entropy using ABC and LattE.

Numeric implementation is significantly faster, with ABC String second fastest, and the Latte implementation slowest.

However, I do not claim that ABC is faster than Latte. ABC is implemented as a shared library in SPF allowing for direct function calls to the model counter. On the other hand, in order for SPF to pass constraints to Latte, they are first preprocessed and simplified using the Omega library and then saved to a set of files. Latte is then invoked on these files and the model counts are parsed back into JPF. In order to make any claims about the relative efficiency of ABC and LattE one would need to do a comparison of the constraint model counting capabilities directly. This is future work.

The remaining experiments were conducted using ABC Numeric as the model counter, due to the relative execution speed of the implementation within SPF.

4.5.2 Single- and Multi-run Symbolic Execution

As described in Section 4, I have given two methods for computing the entropy after the adversary makes k observations: performing symbolic execution over the k-composition of the program under an adversary model (Section 4.1.1) and performing

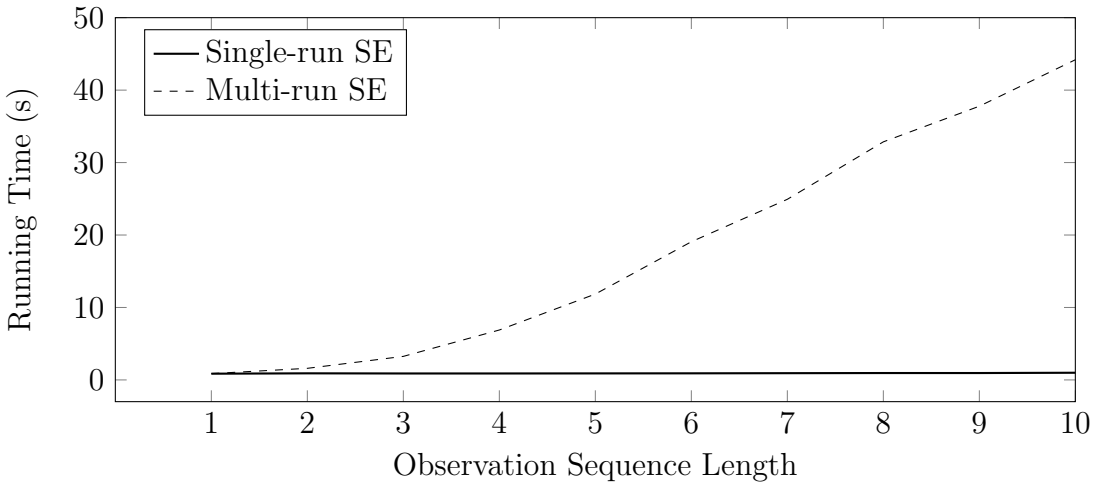


Figure 4.5: Time for multi-run and single-run symbolic execution.

symbolic execution over a single copy of the program and then using mathematical formula to infer the multi-run entropies (Section 4.2).

We ran both analyses on the password checking example and, as expected, we see in Figure 4.5 that the multi-run analysis takes much longer, due to the exploration of many more paths generated by symbolic execution. Both analyses give the same results.

4.5.3 Password Checker

I also present results on the timing analysis of the password checking function. I present results only for multi-run analysis using single-run execution here, as we have just described that it is much faster and produces the same results. I first describe results for a small configuration where we fix the alphabet size to 4 and the password length to 3. Let us assume that the adversary can make k guesses, and let us compute the remaining entropy and the information leakage as shown in Figure 4.6. There are $4^3 = 64$ possible inputs for h giving $\log_2 64 = 6$ bits for the initial entropy. As the adversary makes more guesses, the remaining entropy decreases from 6 to 0. Indeed, the analysis shows that the entropy is 0 for $k \geq 10$. Symmetrically, one can see that the information leakage

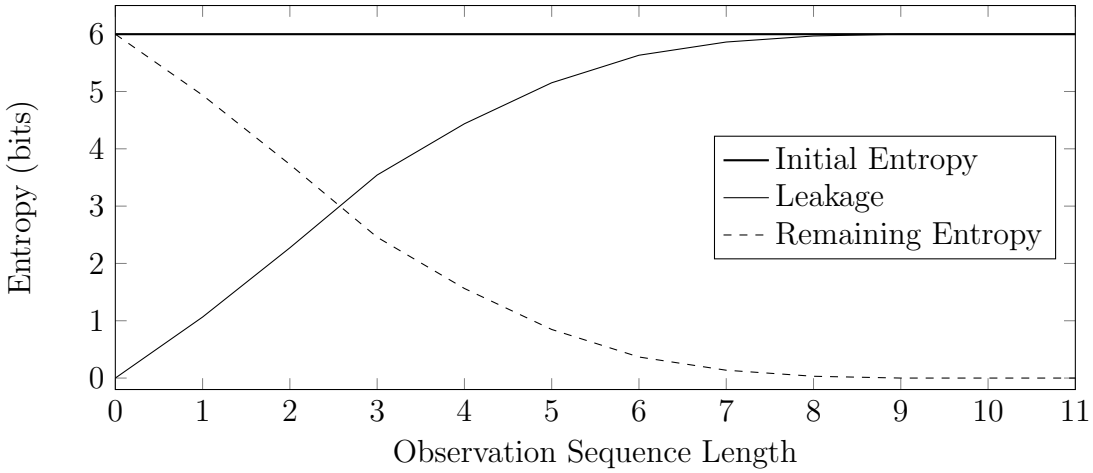


Figure 4.6: Information leakage and remaining entropy for password checking function.

increases with more guesses, from 0 to 6, indicating that all information about the secret is leaked after 10 guesses. Thus, we conclude that the adversary needs at most 10 guesses to fully determine the secret.

I also analyzed a larger configuration. For a password of length 10 and an alphabet of size 128, I incrementally increased the guessing budget of the adversary and determined that 15 guesses are required to reveal 1 bit of information. This analysis took 135.34 seconds.

4.5.4 Text Concatenation and Compression

I further analyzed side channels that depend on the size of the output. One example of such an attack is known as "Compression Ratio Info-leak Made Easy" (CRIME)[20]. The function `concatAndCompress()` shown in Figure 4.7 accepts an input `low` which is controlled by the adversary, concatenates it with a secret value `high`, and then uses the Lempel-Ziv (LZ77) [1] compression algorithm on the resulting string. The code for `LZ77compress` is shown in Figure 4.8.

The basic idea behind the attack is that if the adversary provides a value for `low` that

```
1 public concatAndCompress (String low){
2   return LZ77compress(high.concat(low));
3 }
```

Figure 4.7: A function with a size-based side channel.

does not have a common prefix with `high`, then there will be little compression. However, if `low` and `high` do share a prefix, this will result in a higher compression ratio. This is real-world vulnerability that can be used to reveal secret web session tokens to a malicious user by observing compressed network packet size [19]. Such a user is able to control input through, say, a web form, which is later concatenated with session information and sent to the server. For instance, suppose the secret value `high` is the text `sessionkey:xb5du`. If the malicious user sets the value of `low` to be the text string `sessionkey:abcde` he will observe less compression than if he sets `low` to be `sessionkey:xb5da`. In this way, the attacker is able to make repeated guesses and incrementally learn more information about prefixes of the secret. Thus, the `concatAndCompress()` function acts as a segmented oracle with a side channel on the size of the output.

We applied the analysis to `concatAndCompress()` and were able to compute the information leakage for a given budget on the number of guesses used by the adversary. Due to the complexity of the LZ77 algorithm, symbolic execution becomes more expensive than in the case of the password checking function. For a secret of length 3 and alphabet size 4 single-run symbolic execution generates 187 path conditions leading to 4 different observables. For each observable o_i, we built the disjunction of all the PCs that result in o_i and we used Z3 to prove logical equivalence to the PC formulation in Section 4.2. Using the single-run method we then determined that the `concatAndCompress()` function leaks all information about the secret after 10 executions by the adversary. Using ABC Numeric for model counting, the total running time of this analysis is 8.695 seconds. We repeated this experiment using ABC String as the model counter. The same results

```
1  public static byte[] compress(final byte[] in) throws IOException {
2
3    StringBuffer mSearchBuffer = new StringBuffer(1024);
4    String result = "";
5
6    String currentMatch = "";
7    int matchIndex = 0;
8    int tempIndex = 0;
9    int nextChar;
10   for (int i = 0; i < in.length; i++){
11     nextChar = in[i];
12
13     tempIndex = mSearchBuffer.indexOf(currentMatch + (char)nextChar);
14     if (tempIndex != -1) {
15       currentMatch += (char)nextChar;
16       matchIndex = tempIndex;
17     }
18     else {
19       final String codedString = new StringBuilder().append("~")
20           .append(matchIndex).append("~").append(currentMatch.length())
21           .append("~").append((char)nextChar).toString();
22       final String concat = currentMatch + (char)nextChar;
23       if (codedString.length() <= concat.length()) {
24         result = result + codedString;
25         mSearchBuffer.append(concat);
26         currentMatch = "";
27         matchIndex = 0;
28       }
29       else {
30         for (currentMatch = concat, matchIndex = -1;
31              currentMatch.length() > 1 && matchIndex == -1;
32              currentMatch = currentMatch.substring(1, currentMatch.length()),
33                matchIndex = mSearchBuffer.indexOf(currentMatch)) {
34           result=result+currentMatch.charAt(0);
35           mSearchBuffer.append(currentMatch.charAt(0));
36         }
37       }
38       if (mSearchBuffer.length() <= 1024) {
39         continue;
40       }
41       mSearchBuffer = mSearchBuffer.delete(0, mSearchBuffer.length() - 1024);
42     }
43   }
44   if (matchIndex != -1) {
45     final String codedString = new StringBuilder().append("~").append(matchIndex)
46                                .append("~").append(currentMatch.length()).toString();
47     if (codedString.length() <= currentMatch.length()) {
48       result = result + new StringBuilder().append("~").append(matchIndex).append("~")
49                                .append(currentMatch.length()).toString();
50     }
51     else {
52       result = result + currentMatch;
53     }
54   }
55   final byte[] bytes = result.getBytes();
56   return bytes;
57 }
```

Figure 4.8: The implementation of the the Lempel-Ziv algorithm (LZ77) [1], used in the CRIME case study.

took 152.332 seconds to compute, due to the complex nature of the string operations contained in the LZ77 compression algorithm.

4.6 Chapter Summary

I presented a symbolic execution approach for side channel analysis. The approach quantifies the leakage of the secret information through side channels, which is achieved by computing path probabilities using model counting over symbolic constraints. I illustrated this approach on side channels with segmented oracles and I gave an efficient computation of leakage over multiple attack steps. This technique leverages satisfiability checking and model counting over complex constraints involving both string and numeric operations. In Chapter 6, I extend this side-channel analysis with segmented oracles in the presence of noisy observations and show how to synthesize segment oracle attacks in this setting.

Chapter 5

Offline Adaptive Attack Synthesis

In this chapter I present symbolic analysis techniques for detecting vulnerabilities that are due to adaptive side-channel attacks and synthesizing inputs that exploit the identified vulnerabilities. I start with a *symbolic attack model* that encodes succinctly all the side-channel attacks that an adversary can make. Using symbolic execution over this model, I generate a set of mathematical constraints where each constraint characterizes the set of secret values that lead to the same *sequence* of side-channel measurements. I then compute the optimal attack, i.e, the attack that yields maximum leakage over the secret, by solving an optimization problem over the computed constraints. I use information-theoretic concepts such as channel capacity and Shannon entropy to quantify the leakage over multiple runs in the attack where the measurements over the side channels form the *observations* that an adversary can use to try to infer the secret. I also propose greedy heuristics that generate the attack by exploring a portion of the symbolic attack model in each step. This technique was implemented in Symbolic PathFinder and applied to Java programs, demonstrating how to synthesize optimal side-channel attacks.

5.1 Multi-Run Adaptive Attacks

Let $P(H, L)$ be a *deterministic* program, where H denotes the high input (secret) and L the low input (public). Similar to previous work [82], Let us assume that the attacker can make one side-channel observation at a time and that there are no errors in the measurements. Let us also assume that the attacker knows the implementation of P. These are strong assumptions that are justified since we are interested in computing the "strongest" possible attack.

In general, the attacker can not learn all the secret from only one round of observation. We are interested in computing the low values for estimating the maximum leakage after the attacker runs the program multiple times and makes multiple side-channel observations to *gradually* uncover information on the secret. One can try to infer this maximal leakage based on the computation of leakage on a single run, but this computation would not be accurate, since the attacker can *learn* from previous tries and will try to pick different low values that uncover different information in each round.

5.1.1 Attacker Model

The attacker can be defined as a (partial) function A that takes the history of observations as input and returns the low value l to be used in the next attack step. One can model the interaction, up to k steps, between the attacker and the program as a system $S = (A, P, k, cost(\cdot))$, where the attacker A generates values of l for multiple executions of program P in order to determine the secret h. Parameter $cost(\cdot)$ determines the side-channel observations for program executions. See Algorithm 9.

Let us assume that each path can give only one observable. Our work is done in the context of a project that targets side-channels for Java programs, where this assumption holds. I relax this restriction in Chapter 6. This work is also applicable to more general

Algorithm 8 The k-step Adaptive Attack Model

```
1: procedure S(A, P, k, cost(·))
2:     seq ← ∅
3:     for i from 1 to k do
4:         l ← A(seq)
5:         o ← COST(P(h, l))
6:         seq ← APPEND(seq, o)
```

quantitative information flow analysis where the same assumption holds.

The attacker is *adaptive* as it picks a new low value with each observation made. In contrast, a non-adaptive attacker would only pick one low value at each attack step, *regardless* of the observations made in previous runs, i.e. the attack function A would be a function of attack step i (and the low value would be $l \leftarrow A(i)$). Thus adaptive attacks can be more powerful, since the attacker can pick different low values at same step.

5.1.2 The Attacker's Knowledge

Suppose that the attacker observed o_1, o_2, .. o_k after picking values L_1, L_2 .. L_k and executing the program k times. The initial domain of the secret is D. At each step i, the attacker learns that the secret *leads* to o_i under L_i and revises the domain to contain those secrets that are consistent with this new observation. The attacker successfully reveals the secret when she can deduce the domain to consist of only one value. However, if after an attack step, the revised domain stays the same, the adversary does not learn new information with this low input.

In general, an adaptive attack can lead to different sequences of observations, depending on the secret value. Following [82] let us say that two secrets H and H' are *indistinguishable* under the attack A (written as $H \approx H'$) if for all observation sequences seq, $cost(P(H, A(seq))) = cost(P(H', A(seq)))$. Observe that the indistinguishability relation under attack A forms an equivalence relation on the secret values. Thus the attack

A induces a *partition* on the secret values. Each block in the partition contains the secret values that lead to the same observations, under an attack A. Furthermore, the size of the partition is equal to the number of different k-sized observation sequences produced under A.

Given the system $S = (A, P, k, cost(\cdot))$ one can extend the classical definitions of Channel Capacity and Shannon entropy to reasoning about *sequences* of observations in S (henceforth called k-observables). The channel capacity theorem [83, 84] states that the leakage for a program (in number of bits on the secret) is always less than or equal to the log of the number of possible distinct observations that an attacker can make. The result states in essence that leakage computation reduces to *counting* the number of different observable outputs for the program.

Thus, if running system S for a particular attack A results in $NkObs$ k-observables, the information leaked by P after k runs is:

$$\text{Information leaked after k runs} \leq CC^k(P) = \log_2(NkObs)$$

For deterministic systems, the Shannon entropy gives a measure of the leakage of the side-channel, corresponding also to the observation gain (on the secret) after an observation. Let us extend this result to multiple observations as follows. For each sequence of observations $o_i^k = \langle o_1, o_2, ..o_k \rangle$, let $p(o_i^k)$ denote the probability of observing o_i^k. Then the Shannon entropy is:

$$\mathcal{H}^k(P) = -\sum_{o_i^k} p(o_i^k) \log_2(p(o_i^k)) \tag{5.1}$$

Different attacks A lead to different leakage. The *most powerful attacker* will want to pick the low values that will leak the most information about the secret. In particular we

```
1 int secret;
2 int public_input;
3
4 if(secret >= public_input)
5   ... perform some computation; //cost=1
6 else
7  ... perform some other computation; // cost=2
```

Figure 5.1: Example code with a "binary search" timing side channel.

are interested in *synthesizing* function A that maximizes Channel Capacity or Shannon entropy. Our work generalizes to other information theoretic measures such as computing the probability of guessing the secret or the Min entropy.

Symbolic Attack Tree Example

I illustrate our approach on the example from Fig. 5.1. The program performs two kinds of operations (of costs 1 or 2) according to the branching condition in the code. Assume that the domain of the secret is $D = 1..6$. Consider a 2-step attack that picks low value 4 in the first step, low value 5 after seeing cost $\langle 1 \rangle$ and low value 3 after seeing cost $\langle 2 \rangle$, i.e. $A(\emptyset) = 4, A(\langle 1 \rangle) = 5, A(\langle 2 \rangle) = 3$. Suppose the attacker first observes $\langle 1 \rangle$. She can then deduce that the value of the secret is greater or equal than 4, thus narrowing down the possible values of the secret to $\{4, 5, 6\}$. Running the program again (on $A(\langle 1 \rangle) = 5$) and after observing $\langle 2 \rangle$, she can deduce that the secret is less than 5, narrowing down the possible secret values to $\{4\}$. Thus the attacker is able to fully recover the secret along this path. If on the other hand the second observation is $\langle 1 \rangle$, this means that the secret is greater or equal than 5, so the attacker is able to narrow down the secret to two values $\{5, 6\}$ etc.

Algorithm 9 Symbolic k-step Adaptive Attack Model

```
1: procedure S_sym(P, k, cost(·))
2:     seq ← ∅
3:     h ← MakeSymbolic('h')
4:     for i from 1 to k do
5:         l ← MakeSymbolic(N(seq))
6:         o ← Cost(P(h, l))
7:         seq ← Append(seq, o)
8:     return seq
```

5.2 Symbolic Execution for Attack Synthesis

Let us use symbolic execution, as described in previous chapters, to synthesize the attack that maximizes Channel Capacity and Shannon Entropy. Let us first create a *symbolic* model of the attack scenario described in the previous section where we model the secret using a symbolic variable. Furthermore, since we do not know in advance the low values that give the maximal leakage, let us model them as fresh symbolic variables, as well. The resulting system, S_{sym} is described below:

The code is similar to the procedure S shown in Section 5.1 except that we use directive $makeSymbolic(name)$ to create high and low symbolic values with the specified name. Notably, at each iteration we create a symbolic value for low ($\mathcal{N}(seq)$), whose name is a *function* of observation sequence seq. Intuitively, this mimics the fact that after observing sequence seq, the attacker learns the information about h that is consistent with these observations, and chooses the next low value accordingly. However, instead of fixing an attack A a-priori and choosing concrete low values based on it, we leave the low values symbolic, encoding all the possible concrete values at each attack step.

Running symbolic execution on S_{sym} will generate a set of *symbolic paths* corresponding to the k invocations of program P. These paths represent all the possible attacks of an adaptive adversary up to k runs, since the symbolic values introduced in S_{sym} represent all the possible concrete values (of high and low). Furthermore, the symbolic paths

generated with a symbolic execution of a system represent all the possible concrete paths through that system [21].

For simplicity let us assume that all the paths terminate within the prescribed bound. As this is not always the case in general, in practice one can use a notion of *confidence* (similar to [81]) that quantifies the impact of the execution bound on the quality of the analysis.

Each path π is a composition of paths $\pi_1; \pi_2..\pi_k$, where each π_i is a path in the i-th invocation of P. The *cost* of π is a k-observable, i.e., a sequence of k side-channel measurements made during the attack. Each path has a corresponding path condition, $PC^k(h, \bar{l})$, which in turn is a *conjunction* of k path conditions obtained from single invocations of P (as prescribed by S_{sym}). Here h denotes the symbolic value of the secret while $\bar{l}$ denotes a *tuple* of symbolic low values as created by S_{sym}. Note that there is a one-to-one correspondence between paths and path conditions. Let us write $cost(PC^k)$ to denote the k-observable for the corresponding path.

A value assignment for the symbolic low variables defines a concrete attack. In particular, let $\mathcal{V}$ be a function that assigns to each symbolic low variable a value from low input's domain. We synthesize A by defining, for each low variable of the form $\mathcal{N}(seq)$,

$$A(seq) = \mathcal{V}(\mathcal{N}(seq))$$

Different value assignments result in different attacks. For each concrete attack we can compute the channel capacity and Shannon entropy as follows. Let $\bar{L} = \langle \mathcal{V}(l_1), \mathcal{V}(l_2), ... \rangle$ denote a value assignment for a concrete attack. For each k-observable o_i^k let us build a clause C_i:

$$C_i(h, \bar{L}) = \bigvee_{cost(PC_j^k)=o_i^k} PC_j^k(h, \bar{L})$$

Intuitively each clause characterizes all the secrets that are indistinguishable under the attack defined by $\bar{L}$. Let $\mathcal{C}$ be the set of all satisfiable clauses. The channel capacity is then:

$$CC^k(P) = \log_2(|\mathcal{C}|)$$

Further, let us compute the Shannon entropy using model counting over the clauses to compute the probabilities for each observation. For a uniform distribution over the secret, the probability of observing o_i^k is given by:

$$p(o_i^k) = \frac{\sharp(C_i(h, \bar{L}))}{\sharp D}.$$

Here $\sharp(c)$ denotes the number of solutions, i.e., possible values satisfying the constraint c. This count can be computed with an off-the-shelf model-counting procedure such as Barvinok [57]. Let us use $\sharp D$ to denote the size of the secret domain D assumed to be (possible very large but) finite. Then leakage according to Shannon entropy is defined as follows:

$$\mathcal{H}^k(P) = -\sum_{i=1,m} \frac{\sharp(C_i(h, \bar{L}))}{\sharp D} \log_2 \left(\frac{\sharp(C_i(h, \bar{L}))}{\sharp D} \right)$$

We are interested in finding the assignment $\bar{L}$ of low variables that yield the maximal leakage according to the two formulas above. Intuitively this value assignment will allow us to compute *bounds* on the information leakage that an attacker can achieve. We reduce the problem of finding this value assignment to optimization problems over the set of clauses, as described later in this section.

Consider our running example. The result of symbolically executing S_{sym} for the program is a set of symbolic paths which can be organized in a tree as shown in Figure 5.2 (for $k = 3$). In the tree, nodes depicted with bold rectangles represent "attacker moves", i.e., choosing low values based on the history of observations. Nodes depicted with light rectangles represent "system responses", i.e., the side-channel measurements made along program runs. Let us also depict the intermediate path conditions computed for the different observations. The path conditions encode the constraints on the secret that the attacker is *learning* with each observation, while attempting to narrow down the values of the secret. The path constraints on the leaves represent the knowledge that the attacker has learned after k steps.

The tree is interpreted as follows. In a first step the attacker chooses symbolic value l for low and runs the program once obtaining observations $\langle 1 \rangle$ and $\langle 2 \rangle$, with path conditions $h \geq l$ and $h < l$ respectively. The attacker, then runs the program a second time, using a new symbolic value l_1 if the observed cost is $\langle 1 \rangle$ and a new symbolic value l_2 if the observed cost is $\langle 2 \rangle$. This second execution results in more constraints on the secret, that help the attacker narrow down its values. For example, if the observed cost sequence is $\langle 1, 2 \rangle$, one can determine that the secret satisfies the constraint $h \geq l \wedge h < l_1$.

Note that the tree encodes multiple concrete attacks all at once. For example, consider an attack that in the first run picks low value 4, while in the second run it picks 5 after observing $\langle 1 \rangle$ and 3 after $\langle 2 \rangle$. This corresponds to variable assignment: $l = 4, l_1 = 5, l_2 = 3$. After $\langle 1, 2 \rangle$ the attacker has learned that $h \geq 4 \wedge h < 5$ which narrows down the possible values of h to only one value (4). Consider now a different variable assignment: $l = 4, l_1 = 6, l_2 = 3$. After $\langle 1, 2 \rangle$ the attacker has learned that $h \geq 4 \wedge h < 6$ which narrows down the possible values of h to two (4 and 5) etc.

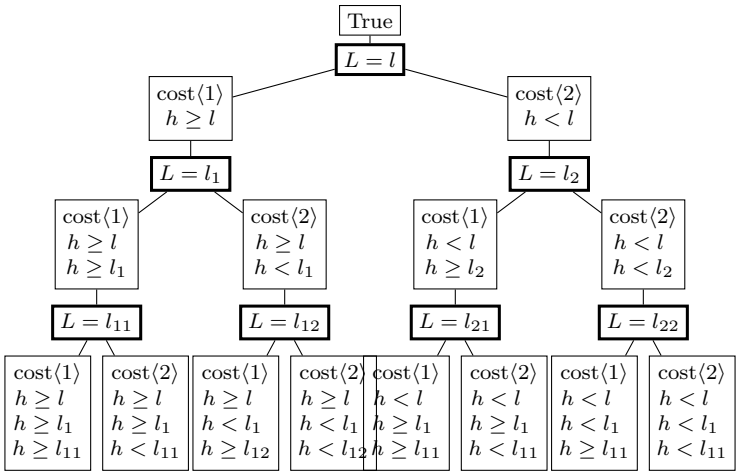


Figure 5.2: Symbolic tree for running example.

5.3 Maximizing Channel Capacity

Let us compute the optimal strategy with respect to channel capacity using MaxSMT solving [85], an extension of the result from [33] which, however, only applied to non-adaptive attacks. MaxSMT [86] is an extension of SMT (satisfiability modulo theories) solving to optimization: given a weighted first-order formula composed of a set of clauses, each with a weight (positive or infinity), MaxSMT finds the assignment that minimizes the sum of the weights of the falsified clauses, or alternatively maximizes the sum of satisfied clauses.

In this setting, consider a set $\mathcal{C} = \{C_1, C_2 \dots C_n\}$ of clauses, where each C_i has the weight 1. The MaxSMT problem is then to find a subset $\mathcal{M} \subseteq C$ with largest cardinality, such that $\mathcal{M}$ is satisfiable.

The approach is illustrated in Algorithm 10. Procedure ComputeConstraints builds a set $\mathcal{C}$ of clauses, where, as before, each clause $C_i(h, \bar{l})$ is a disjunction of the path conditions leading to the same k-observable. Note, however, that the values of low are left symbolic. This set is processed by procedure MaxCC as follows. Let us transform

Algorithm 10 Adaptive Attack Synthesis by Channel Capacity

1: **procedure** AdaptiveMaxLeakCC($P, k, cost(\cdot)$)
2: $\mathcal{C} \leftarrow ComputeConstraints(P, k, cost(\cdot))$
3: $(w, \bar{L}) \leftarrow MaxCC(\mathcal{C})$ **return** $(\log_2 w, \bar{L})$

Algorithm 11 Adaptive Attack Constraint Computation

1: **procedure** ComputeConstraints($P, k, cost(\cdot)$)
2: $\mathcal{O} \leftarrow \emptyset$, $\mathcal{C} \leftarrow \emptyset$
3: $\mathcal{PC} \leftarrow$ SymbolicExecution($S_{sym}(P, k, cost(\cdot))$)
4: **for all** $PC_i^k(h, \bar{l}) \in \mathcal{PC}$ **do**
5: $\mathcal{O} \leftarrow \mathcal{O} \cup \{cost^k(PC_i^k(h, \bar{l}))\}$
6: **for all** $o_i^k \in \mathcal{O}$ **do**
7: $C_i(h, \bar{l}) \leftarrow \bigvee_{cost(PC_j^k)=o_i^k} PC_j^k(h, \bar{l})$
8: $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_i(h, \bar{l})\}$
9: **return** $\mathcal{C}$

$C_i(h, \bar{l})$ into $C_i(h_i, \bar{l})$ by renaming h with fresh h_i in each clause C_i, respectively. The intuition is the same as in [33]: the clauses are renamed to define constraints on low variables, while the high variables are left unconstrained and the goal is to find the low input value that leads to the maximum number of observations for *any value* of the secret.

Applying MaxSMT to the set of renamed clauses will yield the *maximal* number of clauses that are together satisfiable, and thus yield the maximum number of observations possible (up to k), giving maximum leakage in terms of channel capacity. Further, MaxSMT gives a *solution*, i.e., an assignment $\bar{L}$ to symbolic variables $\bar{l}$ that satisfies the maximum satisfiable clauses, meaning that it induces the partitioning on the secret with the maximum number of equivalence indistinguishability classes, and thus it defines the best k-step attack. Algorithm AdaptiveMaxLeakCC computes the k-step adaptive attack that is optimal w.r.t. Channel Capacity.

As an illustration, consider again the running example. The analysis (up to k=3) yields 8 path conditions, corresponding to cost sequences: $\langle 1, 1, 1\rangle$, $\langle 1, 1, 2\rangle$, $\langle 1, 2, 1\rangle$,

Algorithm 12 Channel Capacity Maximization

```
1: procedure MaxCC(C)
2:     C' ← Rename(C)
3:     (w, L̄) ← MaxSMT(C')
4:     return (w, L̄)
```

$\langle 1,2,2\rangle$, $\langle 2,1,1\rangle$, $\langle 2,1,2\rangle$, $\langle 2,2,1\rangle$, $\langle 2,2,2\rangle$. Each symbolic path yields different cost, thus each path condition corresponds to a clause, giving the following clauses (after renaming):

$$h_1 \geq l \wedge h_1 \geq l_1 \wedge h_1 \geq l_{11} \qquad h_2 \geq l \wedge h_2 \geq l_1 \wedge h_2 < l_{11}$$
$$h_3 \geq l \wedge h_3 < l_1 \wedge h_3 \geq l_{12} \qquad h_4 \geq l \wedge h_4 < l_1 \wedge h_4 < l_{12}$$
$$h_5 < l \wedge h_5 \geq l_2 \wedge h_5 \geq l_{21} \qquad h_6 < l \wedge h_6 \geq l_2 \wedge h_6 < l_{21}$$
$$h_7 < l \wedge h_7 < l_2 \wedge h_7 \geq l_{22} \qquad h_8 < l \wedge h_8 < l_2 \wedge h_8 < l_{22}$$

Solving with MaxSMT gives that the maximum number of satisfiable clauses, corresponding to maximum number of observables after 3 steps, is 6, which is equal to the domain of the secret. Thus, no matter what the value of the secret is (in domain 1..6) an attacker can guess it in maximum 3 steps. Furthermore, MaxSMT provides a satisfying assignment to the values in $\bar{l}$: $l = 3, l_1 = 5, l_2 = 2, l_{11} = 6, l_{12} = 4$ defining an attack as illustrated in Figure 5.3. The leaves in the tree define the partition on the secret induced by this attack. All the blocks in the partition have size 1, confirming that the attacker can always guess the secret for this example. Note that the partitions on the right subtree in the figure already have size 1 after two steps. Thus, a third attack step is not necessary – in the implementation one can exploit such situations to perform *pruning* of the attack tree, as explained later in this section.

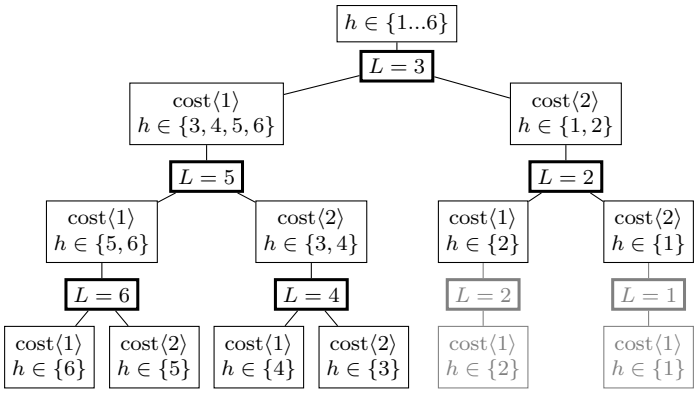


Figure 5.3: Computed attack tree.

5.4 Maximizing Shannon Entropy

Computing the low inputs (i.e., the attack) that maximize the number of observations does not necessarily lead to the optimal attack with respect to Shannon entropy. I propose alternative strategies that aim to maximize Shannon entropy instead of simply the number of observations.

As described earlier, an attack consists of an assignment of concrete values to symbolic low inputs, $\bar{L} = \langle \mathcal{V}(l_1), \mathcal{V}(l_2), ... \rangle$. Our goal is to choose $\bar{L}$ which maximizes the Shannon entropy given that choice of $\bar{L}$, which I denote $\mathcal{H}^k(P|\bar{L})$. Maximizing this entropy thereby maximizes the expected information leakage after k steps. To achieve this I developed two different methods, MaxHMarco and MaxHNumeric, which are both phrased as combinatorial optimization problems over $\bar{l}$ with objective function $\mathcal{H}^k(P|\bar{l})$. These two methods are complementary: MaxHMarco is guaranteed to return the partition with highest entropy but it is sensitive to the size of the input domain; MaxHNumeric uses numeric optimization methods that are approximate and therefore can not provide full guarantees but they can potentially scale to larger input domains.

Algorithm 13, AdaptiveMaxLeakH, outlines the approach. We first use symbolic execution to compute the set of clauses $\mathcal{C}$ which partitions the input. I then compute

Algorithm 13 Shannon Entropy Maximization

```
1: procedure AdaptiveMaxLeakH(P, k, cost)
2:     C ← ComputeConstraints(P, k, cost(·))
3:     L̄ ← MaxH(C, true)
4:     return (H^k(P|L̄), L̄)
```

the value of $\bar{L}$ which maximizes the entropy by setting the function MaxH to be either MaxHMarco or MaxHNumeric, and finally return the maximum entropy value, which gives the information leakage. The two methods are detailed in the subsequent sections.

5.4.1 Entropy Maximization: Numeric Optimization

Our approach generates a symbolic entropy function and attempts to directly maximize that function using numeric techniques. This method relies on parameterizing observation sequence probabilities by the choice of low input values, computed via model counting.

A model counting function for $C_i(h, \bar{l})$ is a function $F_i(\bar{l})$ that computes the number of possible secrets h that satisfy C_i, given a choice of $\bar{l}$. For instance, recall the running example from Fig. 5.1 with a secret domain $1 \leq h \leq 6$ and suppose one is interested in an attack for 2 steps. The adversary will input an initial guess l, and then input l_1 or l_2 depending on if cost $\langle 1 \rangle$ or cost $\langle 2 \rangle$ is observed. Then there are 4 possible constraints over the vector $\bar{l} = \langle l, l_1, l_2 \rangle$ corresponding to the leaves of the symbolic attack tree.

$$C_1 = h < l \wedge h < l_1 \qquad C_2 = h < l \wedge h \geq l_1$$
$$C_3 = h \geq l \wedge h < l_2 \qquad C_4 = h \geq l \wedge h \geq l_2$$

Consider the number of secrets h that are consistent with C_1 for a given choice of $\bar{l}$ and the domain of h. If both $l > 6$ and $l_1 > 6$ then h can take on any value in the

domain and there are 6 solutions. If $1 \leq l \leq 6 \wedge l \leq l_1$ then there are exactly $l-1$ values of h that satisfy C_1. Symmetrically, if $1 \leq l_1 \leq 6 \wedge l_1 < l$ then there are $l_1 - 1$ possible values for h. Otherwise, C_1 has no solutions. One can write a counting function for this and the 3 remaining constraints as piecewise functions (where it is assumed that if none of the piecewise conditions apply then the function is 0.)

$$F_1(\bar{l}) = \begin{cases} 6 & : l > 6 \wedge l_1 > 6 \\ l-1 & : 1 \leq l \leq 6 \wedge l \leq l_1 \\ l_1 - 1 & : 1 \leq l_1 \leq 6 \wedge l_1 < l \end{cases} \qquad F_2(\bar{l}) = \begin{cases} 6 & : l_1 < 1 \wedge 6 < l \\ l - l_1 & : 1 \leq l_1 \leq l \leq 6 \\ l-1 & : l_1 < 1 \leq l \leq 6 \\ 7 - l_1 & : 1 \leq l_1 \leq 6 < l \end{cases}$$

$$F_3(\bar{l}) = \begin{cases} 6 & : l < 1 \wedge 6 < l_2 \\ l_2 - l & : 1 \leq l \leq l_2 \leq 6 \\ l_2 - 1 & : l < 1 \leq l_2 \leq 6 \\ 7 - l & : 1 \leq l \leq 6 < l_2 \end{cases} \qquad F_4(\bar{l}) = \begin{cases} 6 & : l < 1 \wedge l_2 < 1 \\ 7 - l & : 1 \leq l \leq 6 \wedge l_2 < l \\ 7 - l_2 & : 1 \leq l_2 \leq 6 \wedge l \leq l_2 \end{cases}$$

I use the parameterized model counter Barvinok [57] to automatically produce each $F_i(\bar{l})$. Barvinok performs parameterized model counting by representing a constraint C on variables $\bar{l}$ and h as a symbolic polytope $Q \subseteq \mathbb{R}^n$. Barvinok's algorithm generates a multivariate piecewise polynomial F such that $F(\bar{l})$ evaluates to the number of assignments of integer values to h that lie in the interior of Q.

Using each $F_i(\bar{l})$ one can compute the probability of an observation sequence given the values of the low inputs as $p(o_i^k|\bar{l}) = F_i(\bar{l})/\#D$. One can then plug these symbolic probability functions into Equation 6.1:

Algorithm 14 Symbolic-Numeric Shannon Entropy Maximization

1: **procedure** MaxHNumeric($\mathcal{C}, D$)
2: **for all** $C_i \in \mathcal{C}$ **do**
3: $F_i(\bar{l}) \leftarrow$ SymbolicModelCount($C_i, \bar{l}, h$)
4: $p(o_i^k|\bar{l}) \leftarrow F_i(\bar{l})/\#D$
5: $\mathcal{H}^k(P|\bar{l}) \leftarrow -\sum_{i=1}^{m} p(o_i^k|\bar{l}) \log_2(p(o_i^k|\bar{l}))$
6: $\bar{L} =$ NMaximize($\mathcal{H}^k(P|\bar{l})$)
7: **return** $\bar{L}$

$$\mathcal{H}^k(P|\bar{l}) = -\sum_{i=1}^{m} \frac{F_i(\bar{l})}{\#D} \log_2 \frac{F_i(\bar{l})}{\#D}$$

Then, the attack synthesis can be stated as a non-linear objective function maximization problem, defined by $\bar{L} = \arg\max_{\bar{l}} \mathcal{H}^k(P|\bar{l})$. I leverage existing non-linear optimization routines to approximate $\bar{L}$. In the implementation Mathematica's NMaximize was used. The overall strategy generation algorithm using numeric entropy maximization is as follows:

For the example from Figure 5.1, maximizing $\mathcal{H}(P|\bar{l})$ results in the assignment $\bar{L} = \langle 4, 2, 5\rangle$ for symbolic inputs $\langle l, l_1, l_2\rangle$. This corresponds to the first two steps of an adaptive timing side channel binary search attack.

Note that method MaxHNumeric can also be used, with minimal modifications, for computing an attack with respect to other measures. Once we have the (parameterized) probability computations (line 3) one can plug them in the formulas for, e.g., guessability or Min entropy, and apply numeric optimizations to maximize those measures.

5.4.2 Entropy Maximization: Maximal Satisfiable Subsets

The MaxSMT solution described in the earlier section represents only one maximal partition on the secret which may not necessarily lead to maximum entropy. One can

compute *all* the maximal partitions on the secret and then choose the one that has maximal entropy using the MARCO algorithm [85]. The MARCO algorithm solves a generalization of MaxSMT, namely the problem of finding *all* Maximal Satisfiable Subsets (MSSs) of clauses that are together satisfiable [87]. The details of using the MARCO algorithm for entropy maximization are beyond the scope of this dissertation, but I compare the results of this method with our method for numeric entropy maximization, described in the previous section.

5.4.3 Greedy Maximization

Generating the symbolic attack tree fully up to depth k will generate up to m^k knowledge states, where m is the number of observables. Thus, the maximization methods presented so far would involve over m^k low variables. Rather than perform the full exploration up to a given depth, one can use a d-greedy approach, in which the attack is computed in phases of size d and the l-variables are solved to maximize channel capacity or entropy for each phase. This reduces the problem to solving $\frac{k}{d}$ maximization problems of size m^d, with the trade off of (possibly) yielding a suboptimal solution. Note that in the case of a binary search oracle side channel, as in the example, the 1-greedy solution using Shannon entropy happens to be the optimal solution, requiring solving k optimizations problems each with 1 free parameter, rather than 1 optimization problem with 2^k parameters. In general a greedy solution can be arbitrarily suboptimal [82].

5.4.4 Optimizations

Procedure ComputeConstraints$(P, k, cost(\cdot))$ is used to build a set of clauses, each one corresponding to a k-observable. The simplest and most intuitive implementation is to run Symbolic Execution on the system S_{sym}. However, this implementation can be

optimized by running Symbolic Execution on only one copy of the program to obtain a set of path conditions. One then obtains the clauses corresponding to k-observables by systematically combining these path conditions (with appropriate renaming). This optimization reduces the overhead of symbolically executing the program multiple times. Further, for the greedy techniques early pruning was implemented to not expand attack steps for partition blocks that already have size 1, since no new information can be inferred for those blocks.

5.5 Implementation

The proposed techniques were implmented in the Symbolic PathFinder (SPF) [30] symbolic execution tool. SPF implements a custom JVM which symbolically executes Java bytecode instructions. I use Barvinok [57] for (parameterized) model counting (for linear constraints). For numeric maximization I use MATHEMATICA's NMAXIMIZE function [88] configured to use *Differential Evolution* [89] and set to use between 100 and 250 iterations to balance running time and convergence.

Cost Models

Our work is done in the context of a project that specifically addresses side-channels that are related to time and space consumption in Java programs. To this end, SPF listeners were implemented to monitor the bytecode instructions executed by the program, and to perform the analysis of side-channels related to time and space consumption. For timing channels one can compute the execution time of each (symbolic) path by assigning a time unit to each instruction and aggregating the cost.

To obtain a more realistic cost model, one can also perform statistical measurements of the execution time of the program ran on a reference hardware platform, as driven by

the tests that satisfy the corresponding path conditions. Analysis of other types of side channels can be implemented easily, e.g. one can monitor the memory allocated inside SPF's custom JVM to measure memory consumption.

An abstraction layer was also implemented that groups together the costs that have very close values, as in practice they would be indistinguishable to an adversary. Let o_{min} and o_{max} be the minimum and maximum values of the costs observed along the paths in one run. The range is divided into n intervals from 0^{th} to $(n-1)^{th}$, where n is a user supplied parameter. Each interval has equal size, $d = \frac{o_{max} - o_{min}}{n}$. One can then map all the costs *obs* such that $o_{min} + i \times d \leq obs < o_{min} + (i+1) \times d$ to the same interval i (o_{max} belongs to the $(n-1)^{th}$ interval). These intervals form the *abstractions* of the concrete costs and they are used in the analysis. In practice this abstraction can be used to find optimal attacks, while also providing the benefit of greater scalability (since the number of observations and hence of clauses can be reduced significantly).

5.6 Experiments

There are three main techniques (for a depth k): (1) MaxCC, (2) MaxHMarco, and (3) MaxHNumeric, each with two variants: (a) full exploration and (b) 1-greedy approach. When evaluated, both MaxCC (1a) and (1b) generate effective attacks in reasonable time which is described in the coming sections. For MaxHMarco and MaxHNumeric, I find that due to the complexity of composed constraints, (2a) and (3a) are not feasible in practice. In addition, (2b) and (3b) give optimal or near-optimal attacks. Thus let us consider the four variants (1a, 1b, 2b, 3b) with the goal of assessing if they can automatically synthesize attacks and compute leakage for complex, realistic applications.

LawDB—a complex network service application provided to us by DARPA [90]—and the example of Figure 5.1 were analyzed.

The results of the experiments for MaxCC (1a and 1b) are shown in Fig. 5.1, while the results of the experiments for MaxHMarco and MaxHNumeric (2b and 3b) are shown in Fig. 5.2. In the tables, `DOMAIN` is the number of possible values of the secret. The tables show the number of attack steps, the maximum number of observables, maxObs, the leakage and the analysis time (in seconds). A '-' indicates timeout (of 1h). All the experiments were run on a standard MacBook Pro. I first give a high level description of the discovered vulnerabilities and then describe in more detail the results displayed in the tables.

Vulnerabilities

LawDB is a network service application that provides access to records about law enforcement personnel. The application consists of 41 classes with 2844 line of codes, and uses the Netty library [91]. On the server side, LawDB stores all employee records in a database, and each employee is referenced with a unique ID. All IDs are loaded into a tree data structure when the server starts. There is a group of employees who works on clandestine activities; their IDs are restricted information. On the client side, there are several available operations, including a search for all IDs within a chosen range. Upon receiving the search request, the server delegates it to the tree data structure, which returns all the IDs in the range. If the ID is non-restricted, it is sent back to the client immediately in a UDP package; otherwise the server writes to an error log file, and does not send the restricted ID to the user. To analyze this example the database was populated with two concrete unrestricted IDs and one symbolic restricted ID, i.e. the secret h. The adversary performs the search operation by providing a symbolic range $[l_{min}, l_{max}]$.

Our techniques found a timing channel that is due to the fact that the response time of the server is noticeably longer when there are restricted IDs in the search range

(due to the writing to the log file). Exploiting this timing channel, an adversary can perform an adaptive attack to discover a restricted ID. At a high level, this example is similar to our running example as the optimal attack involves narrowing down a *range* of secret values using repeated comparisons with low values. Specifically, the adversary makes a range request [min, max]. If the secret is in the range, then the execution time is longer. If the secret is outside the range then the time is slower. The adversary keeps making range queries smaller and smaller until it gets to size 1. Our techniques found this attack automatically. Note that for this example an abstraction was used for the costs. First a symbolic analysis was performed on one run of the program and we obtained 30 path conditions and 29 observables. We solved the path conditions, we obtained concrete test inputs and we executed the program (multiple times) on these inputs. Realtime measurements showed that only two group of observables are noticeably different. Therefore, we used abstraction to divide the costs into two intervals obtaining a binary search attack, which we validated by demonstrating it in operation.

Results for MaxCC

The full approach, when it can finish, returns optimal attacks. However, it is expensive, since each analyzed clause encodes what amounts to k copies of path conditions obtained from a single program run. The greedy approach scales better but may not be optimal. See, e.g., results for running example from Fig. 5.1. At each attack step, the adversary provides an input, and can determine from the observation whether the secret is greater or smaller than her input. An optimal attack is a binary search in the secret's domain, which requires $\log_2(\texttt{DOMAIN})$ number of steps in the worst case. Fig. 5.1 confirms that, when $\texttt{DOMAIN} = 10$, the full approach reveals the whole secret in 4 steps ($\log_2(10) = 3.3$, note also that $maxObs = \texttt{DOMAIN}$ so full secret is revealed). In general, a k-step attack would reveal 2^k observables or the whole secret if its domain is less than

2^k. The attacks synthesized by the greedy approach are not optimal. For example, when `DOMAIN` = 200 the optimal strategy requires 8 steps to discover the whole secret; the greedy strategy requires 16 steps. However, it can synthesize this 16-step attack in less than 1 minute, while the full approach times out.

Note also that with the same number of steps, the full approach times out in small domains (200 - 500), but returns quickly when the domain is large (10^6). The reason is that when the domain is small some of the clauses are unsatisfiable, and UNSAT instances are usually expensive.

On the other hand, for `LawDB` and the illustrative example, MaxCC greedy does not generate the optimal attack, but still scales well and generates tight bounds on the leakage within a small number of steps as compared to the optimal attacks (see discussion in the next section).

Results for MaxHMarco and MaxHNumeric – greedy

Computing entropy is more expensive than computing channel capacity. In the results, observe that MaxCC does not generate the optimal attack for LawDB and the running example. Thus we apply the MaxH methods to these two examples using a 1-greedy configuration. Results are shown in Fig. 5.2.

In the illustrative example, MaxHMarco can synthesize the optimal strategy for DOMAIN up to 500, where MaxCC timed out, and MaxCC greedy is not able to synthesize the optimal attack. Furthermore MaxHMarco generates an attack for LawDB for a small domain where 7 steps are enough to reveal all 98 secret values. MaxCC is not able to analyze more than 4 steps, and MaxCC greedy needs 17 steps to reveal the full secret.

MaxHMarco relies on enumeration of partitions, so when there is a different partition for each public input it does not scale to large domains, and times out for a domain size of 10^6. On the other hand MaxHNumeric scales well and discovers attacks which leak only

Case Study	DOMAIN	Steps	Full			1-greedy		
			maxObs	CC	time	maxObs	CC	time
Example from Fig. 5.1	10	4	10	3.322	3.060	8	3	2.330
	200	8	-	-	-	64	6	10.026
		16	-	-	-	200	7.644	27.225
	300	9	-	-	-	83	6.375	11.561
		21	-	-	-	300	8.229	36.827
	400	9	-	-	-	97	6.600	13.143
		20	-	-	-	400	8.644	49.103
	500	9	-	-	-	71	6.150	10.049
		25	-	-	-	500	8.966	1m1.076
	10^6	10	1024	10	14.578	461	8.849	1m0.702
		11	2048	11	2m2.680	752	9.555	1m40.382
		12	4096	12	19m32.370	1199	10.228	2m39.689
		13	-	-	-	1903	10.894	4m15.845
LawDB	100 - 2	2	4	2	2.750	4	2	2.441
		3	8	3	18.915	7	2.807	3.244
		4	16	4	3m8.783	11	3.459	4.160
		5	-	-	-	17	4.087	5.342
		17	-	-	-	98	6.615	21.010
	10^6 - 2	2	4	2	2.762	4	2	2.579
		3	8	3	19.493	8	3	3.866
		4	16	4	3m21.688	16	4	5.844
		5	-	-	-	32	5	9.212
		17	-	-	-	6070	12.567	18m31.246

Table 5.1: Results for MaxCC (full exploration and 1-greedy).

slightly less information for our examples. We also performed experiments for LawDB without abstraction, and found that, as expected, the performance of both MaxHMarco and MaxHNumeric starts to degrade when the number of constraints increases. Thus the role of the abstraction is essential for the analysis of large systems and I plan to investigate it further in the future.

5.7 Chapter Summary

In this chapter, I described techniques for synthesizing adaptive side channel attacks in a fully static, offline manner. I demonstrated the effectiveness of this approach on a set of benchmarks. In the next chapter I will show how to extend this approach to handle programs running on a system that contains noise.

Case Study	**DOMAIN**	**Steps**	**MaxHNumeric**		**MaxHMarco**		
			Leakage (bits)	time	maxObs	Leakage (bits)	time
Example from Fig. 5.1	200	8	7.207	44.876	200	7.644	2m17.120
	300	9	7.560	69.383	300	8.229	3m52.363
	400	9	7.743	1m55.212	400	8.644	5m41.539
	500	9	7.800	1m24.068	500	8.966	7m36.105
	10^6	10	8.172	3m15.000	-	-	-
		11	8.303	4m55.088	-	-	-
		12	8.357	7m12.280	-	-	-
		13	8.371	9m34.512	-	-	-
		14	8.376	12m20.844	-	-	-
LawDB	100 - 2	2	1.999	2.552	4	1.999	1m5.234
		3	2.999	4.688	8	2.999	1m33.656
		4	3.998	10.284	16	3.998	1m49.308
		5	4.996	17.604	32	4.996	2m15.564
		6	5.921	33.852	64	5.921	2m25.816
		7	6.614	57.36	98	6.615	2m36.325
	500 - 2	2	1.999	3.128	4	1.999	6m0.768
		3	2.999	7.340	8	2.999	8m39.441
		4	3.999	10.816	16	3.999	10m33.013
		5	4.999	22.828	32	4.999	12m52.701
		6	5.997	39.844	64	5.997	15m20.654
		7	6.994	1m9.876	128	6.994	15m34.624
		8	7.966	2m6.796	256	7.985	17m43.237
		9	8.760	3m32.292	497	8.955	18m4.668
	10^6 - 2	2	2.	3.652	-	-	-
		3	3.	7.452	-	-	-
		4	4.	13.3	-	-	-
		5	4.999	25.24	-	-	-
		6	5.999	45.544	-	-	-
		7	6.999	1m26.22	-	-	-
		8	7.996	2m41.136	-	-	-
		9	8.939	4m31.396	-	-	-
		10	9.678	8m38.272	-	-	-
		11	10.06	15m8.224	-	-	-

Table 5.2: Results for MaxHNumeric and MaxHMarco.

Chapter 6

Online Adaptive Attack Synthesis Under Noisy Conditions

This chapter contains the primary contribution of my dissertation research. In this chapter, I privide a method for automatically and dynamically synthesizing adaptive side-channel attacks against code segments that manipulate secret data. I synthesize attacks in the presence of noisy environments, like a client and server communicating through a network.

To illustrate the main idea, consider the following scenario. An adversary obtains the program source code and system specification for a server application, but does not have access to private values stored on the server on which the application runs. This is a realistic situation, as many web applications are developed using open-source software and they run on cloud servers with known characteristics. We show that by analyzing the source code and performing offline profiling on a mock server under his control, an adversary can develop a probabilistic model of the relationship between the inputs, side channel measurements, and secret values. Using this model, the adversary mounts a side-channel attack against the real platform, adaptively selecting inputs that incrementally

leak secret values.

In this work, I show how to automate the type of attack just described. I give a widely applicable solution for the problem based on information theory and Bayesian inference. I use state-of-the-art tools and techniques including symbolic execution, symbolic model-counting, and numeric optimization. I implemented this approach, targeting networked Java server applications, and experimentally show that it works on a set of benchmarks by either synthesizing an attack or by reporting that an attack was not found.

There has been prior work: on secure information flow providing methods for detecting insecure flows [92]; on quantitative information flow presenting techniques for measuring the amount of information leaked through indirect flows [37]; and on analysis of adaptive side-channel adversaries providing techniques for automatically reasoning about malicious users [93]. However, despite influential prior work in these areas, existing adaptive adversary models for reasoning about malicious users (1) rely on explicit strategy enumeration via exhaustive approaches [93], (2) attempt to generate an entire strategy tree [85], and (3) do not address environment measurement noise [85]. The contribution of this paper is a novel approach based on symbolic execution [21], weighted model counting [47, 64, 48], and numeric optimization [94] for the online automatic synthesis of attacks that leak maximum amount of private information, and directly addresses the above issues by (1) symbolically representing program behaviors, (2) generating strategy steps dynamically, and (3) using Bayesian inference to model adversary knowledge in a noisy environment.

6.1 Motivating Example

Consider the pseudo-code in Figure 6.1. A client sends a *low security* input l and gets back a response, r. On the other end, a server runs forever waiting to receive l and

Server:

private $h = 97014599$;

private $f(h, l)$:
    if $(h \leq l)$ log.write("bound error");
    else process(l);
    return;

while(true):
    receive l;
    $f(h, l)$;
    send 0;

Client:

send l;
receive r;

Figure 6.1: Example client-server application which contains a side channel.

calls a private function $f(h, l)$, where h is initialized as a private *high security* integer variable representing a secret resource bound, and then responds with 0. The function f compares h and l, writing to an error log if l is too large.

Suppose a malicious adversary $\mathcal{A}$ wants to know the server's secret resource bound h. $\mathcal{A}$ reasons that sending an input which causes the server to write an error to the log should cause a longer round-trip time delay between send and receive than an input which does not cause this error. Now, imagine that the adversary is attacking an idealized system in which this time difference will always be the same, say, for the sake of the example, 2 ms when nothing is written to the log and 4 ms when there is a write to the log. This timing difference gives the adversary a side channel in time which can be exploited to extract the value of h. $\mathcal{A}$ can then try different values of l and decide if the value of h is larger than l or not based on the elapsed time. This enables a binary search on h. In Figure 6.2 we see pseudo-code for such an attack, assuming h is a 32-bit unsigned integer. This is an example of an *adaptive attack*, in which $\mathcal{A}$ makes choices of l based on prior observations. In this paper, we present techniques for *automatically* synthesizing such attacks in the presence of system noise.

Existing works on automated adaptive attack synthesis assume idealized conditions [93,

$min = 0;\ \ max = 2^{32};$
while $(min \neq max)$
  $l = \lceil \frac{min+max}{2} \rceil$
  $t =$ time { send l; receive r; }
  if $(t = 2$ ms$)\ \ min = l;$
  else $max = l;$
$h = min;$

Figure 6.2: Example side-channel attack to reveal the secret value of h stored on the server in Figure 6.1.

85]. However, due to noise in the server and the network, an attack which works for the idealized system is not applicable in practice. The observable elapsed time for each path of f is not a discrete constant value, but follows some probability distribution, thereby obscuring the distinguishability of program paths.

In our example, suppose that, from the client side, the timing measurements from each branch follow distributions approximately centered around 2 ms and 4 ms as in Figure 6.3. If $\mathcal{A}$ sends $l = 100$, observing a 1 ms duration almost certainly implies that $h > 100$ and observing a 5 ms duration almost certainly implies that $h \geq 100$. But, if $\mathcal{A}$ observes a 2.8 ms duration, it appears to be a toss-up—it could be that either $h > 100$ or $h \leq 100$ with nearly equal likelihood.

We present an approach by which an adversary automatically synthesizes inputs to extract information about secret program values despite noise in the system observables. At a high level, we perform offline profiling of a *shadow system* under our control which mimics the real system, in order to estimate the observable probability distributions. Armed with these distributions and some initial belief about the distribution of the secret, we iteratively synthesize a system input l^* which yields the largest expected information gain by solving an information-theoretic objective function maximization problem. In each attack step, the synthesized adversary 1) invokes the system with l^*, 2) makes an observation of the *real* system, 3) makes a Bayesian update on its prior belief about the

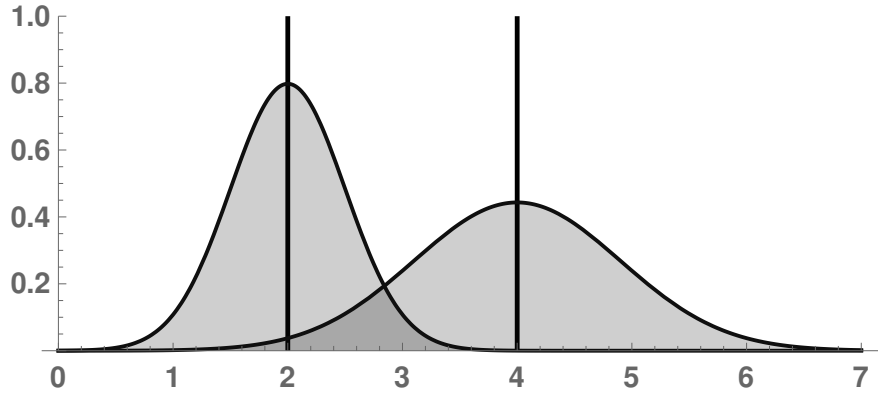


Figure 6.3: Distributions of timing measurements corresponding to the two branches of the server's function f from Figure 6.1: $h \leq l$ (left) and $h > l$ (right).

secret value, and repeats the process for the next attack step.

6.2 Overview

In this section, we give an overview of our model for the adversary-system interaction, make explicit the program parameters over which we conduct our analysis, provide the high-level steps of dynamic attack generation, and give a discussion of relevant information theory concepts.

6.2.1 System Model

In our approach, we analyze secret-manipulating code executing on a server. Note, many side-channel vulnerabilities can be localized to a small number of functions, and it is even very useful to analyze individual functions for vulnerability to side-channel attacks (`memcmp`, `array.equals()`, `String.equals()`) [5, 6, 18, 28, 65]. Once the suspicious function is identified, the attack synthesis is fully automatic. For large systems, this approach can be used to analyze code segments that manipulate the secret, rather than the whole system. n a realistic scenario, a developer or adversary can identify a suspicious

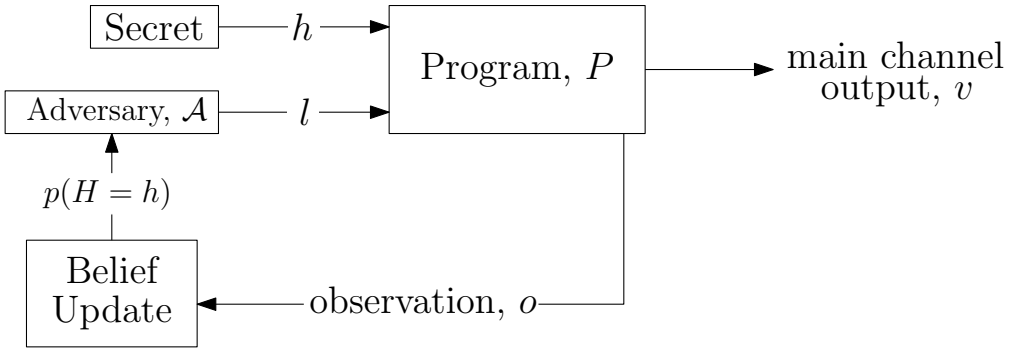


Figure 6.4: Model of adversary, program, inputs, and observations as a probabilistic system.

function that might be vulnerable to side-channel attacks and use our approach to identify such an attack. For the remainder of our discussion we shall assume that P is a program that contains the suspicious function and a direct entry point to invoking the function.

We use a model in which an adversary $\mathcal{A}$ interacts with a system S that is decomposed into a program P and a noise function $\mathcal{N}$, illustrated in Figure 6.4. The program and runtime environment form a probabilistic interactive system acting as an information channel, modeled as the probability of a noisy observation o given the inputs h and l: $p(O = o|H = h, L = l)$. An attacker who wants to learn the secret input h is interested in "reverse engineering" that probabilistic relationship. The adversary wants to compute the probability of a secret input value h given knowledge of his public input l and the side-channel observation o: $p(H = h|O = o, L = l)$. We use Bayesian inference to formalize this process of reverse engineering. By solving a numeric entropy optimization problem at each attack step, an attacker selects optimal input l^* with the goal that $p(H = h)$ converges to a distribution that assigns high probability to the actual initially unknown value of h. Here we define the components of our model.

Program under attack. We assume that P is a deterministic program, which takes exactly two inputs h and l, and we write $P(h, l)$ to indicate the invocation of P on its inputs. The ability of our model to handle programs with more than two inputs is

explained in the definitions of h and l. We explain how our model can be relaxed to handle non-deterministic programs in Section 6.4.7.

High security inputs. By $\mathbb{H}$ we denote the set of possible *high security values*, with h being specific member of $\mathbb{H}$, and H being a random variable which ranges over $\mathbb{H}$. These secret inputs on the real system are not directly accessible to the adversary $\mathcal{A}$ whose goal is to learn as much as possible about the value of $h \in \mathbb{H}$. We assume that the choice of h does not change over repeated invocations of P.

Low security inputs. By $\mathbb{L}$ we denote the set of *low security values*, with l being a specific member of $\mathbb{L}$, and L the corresponding random variable. These inputs are under the control of the adversary $\mathcal{A}$ who chooses a value for l and invokes P with this input. $\mathcal{A}$ may choose different values of l over repeated invocations of P.

Observations. The adversary is able to make side-channel observations, like time measurements, when the program is run. We denote the set of possible observations by $\mathbb{O}$. We assume that the set $\mathbb{O}$ is continuous for the purpose of modeling timing side channels, with O a random variable.

Program traces. A trace t is characterization of a single program execution. We suppose that a run of P according to a trace t is manifested, from $\mathcal{A}$'s point of view, as a side-channel observation o which may be distorted by noise.

The adversary. The adversary $\mathcal{A}$ has some current belief $p(H = h)$ about the value of the secret, chooses an input l to provide to the system, and makes a side-channel observation, o, of the system for that input. $\mathcal{A}$ then makes a Bayesian update on $p(H = h)$ based on o and repeats the process. The contribution of this paper is the synthesis of the optimal inputs, l^*, which cause the system to leak the most information about h.

6.2.2 Outline of Attack Synthesis

Our attack synthesis method is split into two phases. The reader can refer to Figure 6.5 for this discussion.

Phase 1: Offline static analysis and profiling. The main goal of the offline phase is to estimate a probabilistic relationship between program traces and side-channel observations. Informally, a program trace is a sequence of program states, including control flow choices at each branch condition. We group program traces into trace classes based on indistinguishability via observation. We summarize the main points of this phase below, and provide the detailed discussions in Section 6.3.

1. I use symbolic execution to compute path constraints (PCs) on the secret and public inputs for the program source code. Path constraints are logical formulas over inputs that characterize an initial partition estimate for program traces. Each PC, ϕ_i, is associated with a trace class T_i, (Section 6.3.2).

2. For each PC, ϕ_i, I generate a witness $w_i = (h_i, l_i)$. Each w_i is assumed to be a characteristic representative of all concrete secret and public inputs that cause P to execute any of the traces in a trace class T_i.

3. For each w_i, I repeatedly run the system with w_i as input and record observation samples. From the samples, we estimate the conditional probability of observation given trace class, denoted $P(O|T = T_i)$ (Section 6.3.3).

4. PCs may generate too fine a partition of program trace classes. That is, there may be two trace classes T_i and T_j where $P(O|T = T_i)$ and $P(O|T = T_j)$ coincide to such a degree that the traces are effectively indistinguishable by measuring O. Thus, we merge PC's and corresponding distributions which are too similar according to a metric known as the Hellinger distance (Section 6.3.4).

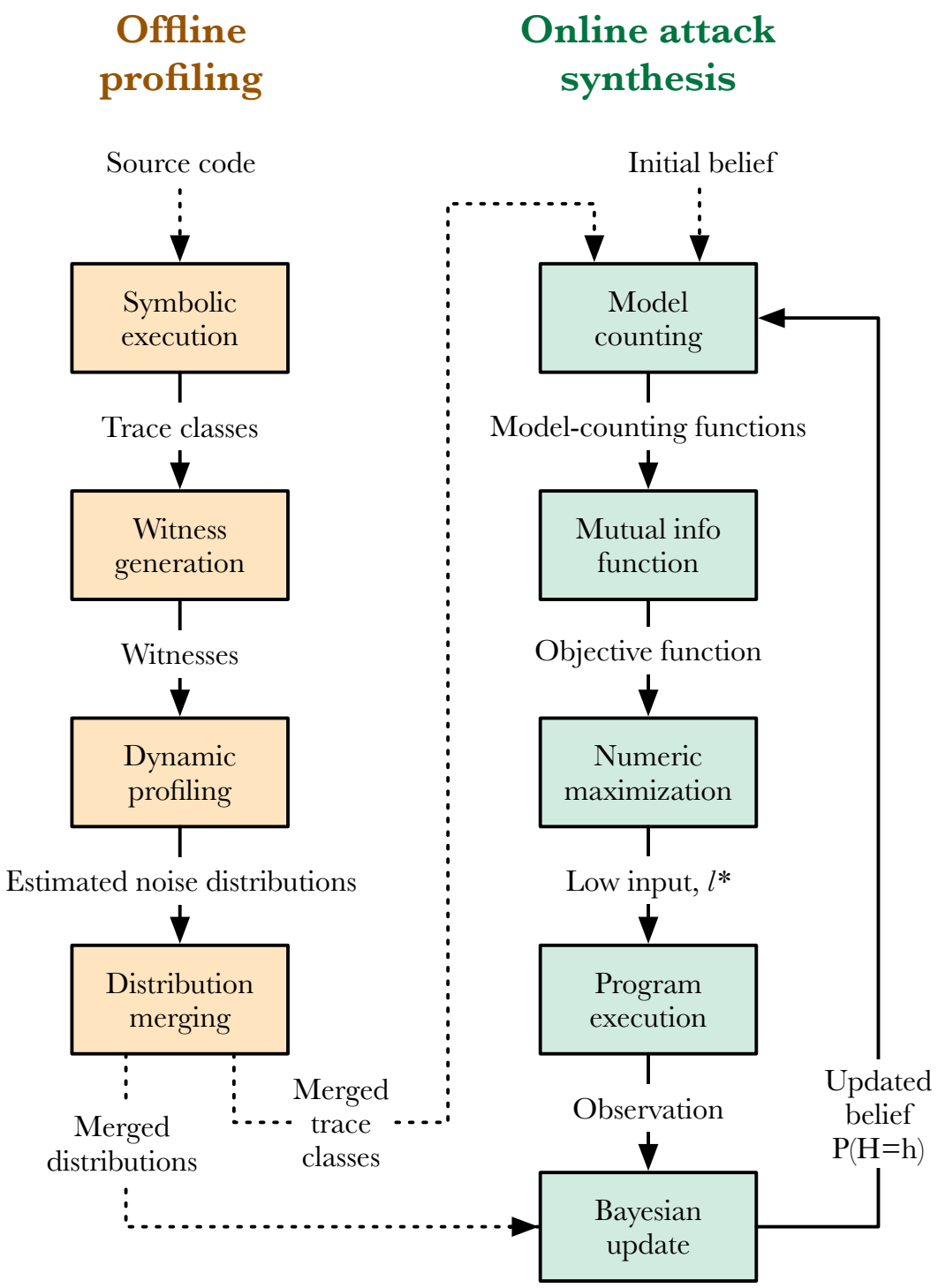


Figure 6.5: Overview of our attack synthesis approach.

Phase 2: Online dynamic attack synthesis. The second phase mounts an adaptive attack against the system, making use of the estimated system profile from the offline phase, assuming that the adversary has some initial belief about the distribution of the secret. We summarize the main approach for this phase below, with detailed discussions in Section 6.4.

1. We use the current belief about the secret along with path constraints to compute, for each ϕ_i, a model counting function which is symbolic over the public inputs, denoted $f_i(l)$(Section 6.4.2).

2. We use the model-counting functions to compute trace class probabilities as symbolic functions over low security inputs. We then apply the mutual information formula from information theory to get an information leakage objective function, which is symbolic over the public inputs (Section 6.4.3).

3. We use numeric optimization to compute the public input l^* that maximizes the symbolic information leakage objective function (Section 6.4.4).

4. We provide the leakage-maximizing input, l^*, to the system and record the observation.

5. We use Bayesian updates to refresh the current belief about the secret using the observation and the noise profile that was estimated during the offline phase (Section 6.4.5). We repeat this process starting from Step 1.

6.2.3 Measuring Uncertainty

In order to maximize information leakage we must have a way to quantify it. Here we give a brief background on standard measures from information theory that are commonly used in side-channel analysis. Intuitively, before invoking P, $\mathcal{A}$ has some *initial*

uncertainty about the value of h, while after observing o, some amount of *information is leaked*, thereby reducing $\mathcal{A}$'s *remaining uncertainty* about h. The field of quantitative information flow (QIF) [37] formalizes this intuition by casting the problem in the language of information theory using *Shannon's information entropy* which can be considered a measurement of uncertainty [38]. We briefly give three relevant information entropy measures [39]. Given a random variable X which can take values in $\mathbb{X}$ with probabilities $p(X = x)$, the *information entropy* of X, denoted $\mathcal{H}(X)$ is given by

$$\mathcal{H}(X) = \sum_{x \in \mathbb{X}} p(X = x) \log_2 \frac{1}{p(X = x)} \tag{6.1}$$

Given another random variable Y and a conditional probability distribution $p(X = x|Y = y)$, the *conditional entropy of X given Y* is

$$\mathcal{H}(X|Y) = \sum_{y \in \mathbb{Y}} p(Y = y) \sum_{x \in \mathbb{X}} p(X = x|Y = y) \log_2 \frac{1}{p(X = x|Y = y)} \tag{6.2}$$

Intuitively, $\mathcal{H}(X|Y)$ is the expected information contained in X given knowledge of Y. Given these two definitions, we would like to compute the expected information gained about X by observing Y. In our application, we target timing side channels where we model time as a continuous random variable, while the secret is a discrete value (i.e., an integer or a string). In order to measure the mutual information between a discrete random variable Y and a continuous random variable X, we use the Kullback–Leibler (KL) divergence [39].

The KL divergence, $D_{\mathrm{KL}}(p, q)$ is a statistical measure of the discrepancy between two models, $p(x)$ and $q(x)$, for a probabilistic event X over a continuous domain. It is

computed via the formula:

$$D_{\mathrm{KL}}(p, q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} \, \mathrm{d}x. \tag{6.3}$$

Then, the *mutual information* between a discrete random variable Y and a continuous random variable X is defined as the expected information gain with expectation taken over all possible events in Y:

$$\mathcal{I}(Y; X) = \sum_{y \in Y} p(Y = y) D_{\mathrm{KL}}(p(X|Y = y), p(X)) \tag{6.4}$$

Intuitively, $D_{\mathrm{KL}}(p(X|Y = y), p(X))$ is a measure of information gain between the prior distribution $p(X)$ and the posterior distribution $p(X|Y = y)$.

We consider the high-security input, low-security input, and observable as random variables H, L, and O, where H and L are discrete and O is continuous. We interpret $p(H)$ as the adversary's initial belief about h and $\mathcal{H}(H)$ as the initial uncertainty. The conditional entropy $\mathcal{H}(H|O, L = l)$ quantifies $\mathcal{A}$'s remaining uncertainty after providing input $L = l$ and observing output O. Finally, we interpret $\mathcal{I}(H; O|L = l)$ as the amount of information leaked.

The goal of the adversary is to maximize the value of $\mathcal{I}(H; O|L = l)$ at every step, which we call the leakage for that step, measured in bits of information. Our attack synthesis technique relies on choosing an optimal input $L = l^*$ at each step which maximizes the expression given in Equation 6.4. At a high level, we compute a symbolic expression for $\mathcal{I}(H; O|L = l)$ using symbolic weighted model counting and then use off-the-shelf numeric maximization routines to choose l^* (detailed in Section 6.4).

6.3 Offline Profiling

The first phase of our attack synthesis relies on offline pre-computation of equivalence classes of program traces and corresponding noise distributions. We accomplish this through the use of symbolic execution, dynamic profiling using a small representative set of witness inputs, and statistical measures for merging indistinguishable distributions. The results of this offline phase are then used during the online attack synthesis phase described in Section 6.4.

6.3.1 Trace Equivalence Classes

Let $\mathbb{T}$ denote the set of execution traces of program P. A trace is a characterization of a single program execution. From adversary $\mathcal{A}$'s point of view, invoking P induces execution of a single trace t. Knowing the input l and the executed program trace t, $\mathcal{A}$ can gain information about h. However, $\mathcal{A}$ does not which program trace t was executed, but can make a side-channel observation $o \in \mathbb{O}$, which may be distorted by system noise. In this setting, there are two challenges:

1. **Observation Noise:** the same execution trace t may lead to different observations o_1 and o_2 in different runs of P.

2. **Observation Collision:** two different traces t_1 and t_2 may lead to the same observation o in some runs of P.

We address these two challenges by defining *trace classes*, which identify observationally equivalent traces in the presence of noise. Let T be a random variable that ranges over $\mathbb{T}$ and $p(O|T)$ be the conditional probability density function on observations conditioned on which trace is executed. We define an equivalence relation $\cong$ on $\mathbb{T}$ in which $t_1 \cong t_2$ if $p(O|T = t_1) = p(O|T = t_2)$ and we say t_1 and t_2 are observationally equivalent.

Let partition $\mathcal{T}$ be the set of equivalence classes of $\mathbb{T}$ defined by $\cong$. Then, we call each $T_i \in \mathcal{T}$ a *trace class*.

$\mathcal{A}$ gains information about h by knowing which trace t was executed when P is run with l. But due to noise and collisions, the best $\mathcal{A}$ can hope for with a single run of the program is to determine the likelihood that $t \in T_i$ for each trace class T_i. So, given l and o, $\mathcal{A}$ would like to know $p(T \in T_i | O = o, L = l)$. In the remainder of this section we show how $\mathcal{A}$ can compute a characterization of $\mathcal{T}$ and efficiently estimate $p(O|T \in T_i)$. We explain in Section 6.4 how $p(O|T \in T_i)$ is used during the online attack phase to compute $p(T \in T_i | O = o, L = l)$.

6.3.2 Trace Class Discovery via Symbolic Execution

I now describe how symbolic execution can be used as a first approximation of trace classes. First, I briefly describe symbolic execution and then explain how symbolic execution's path constraints are associated with trace classes.

Symbolic execution [21], as discussed in Chapter 2, is a static analysis technique by which a program is executed on *symbolic* (as opposed to concrete) input values which represent all possible concrete values. Symbolically executing a program yields a set of *path constraints* $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$. Each ϕ_i is a conjunction of constraints on the symbolic inputs that characterize all concrete inputs that would cause a path to be followed. All the ϕ_i's are disjoint. Whenever symbolic execution hits a branch condition c, both branches are explored and the constraint is updated: ϕ becomes $\phi \wedge c$ in the *true* branch and $\phi \wedge \neg c$ in the *false* branch. Path constraint satisfiability is checked using constraint solvers such as Z3 [29]. If a path constraint is found to be unsatisfiable, that path is no longer analyzed. For a satisfiable path constraint, the solver can return a model (concrete input) that will cause that path to be executed. To deal with loops and

recursion, a bound is typically enforced on exploration depth.

We use the path constraints generated by symbolic execution as an initial characterization of trace classes, where we consider any secret inputs as a vector of *symbolic high security variables* h and remaining inputs as a vector of *symbolic low security variables* l. Then symbolic execution results in a set of path constraints over the domains $\mathbb{H}$ and $\mathbb{L}$, which we write as $\Phi = \{\phi_1(h,l), \phi_2(h,l), \ldots, \phi_n(h,l)\}$. In general, we find it useful to think of each $\phi_i(h,l)$ as a logical formula which returns true or false, but sometimes it is convenient to think of $\phi_i(h,l)$ as a characteristic function that returns 1 when the path constraint is satisfied by h and l and 0 otherwise. A *witness* $w_i = (h_i, l_i)$ for a given ϕ_i is a concrete choice of $h = h_i$ and $l = l_i$ so that $\phi_i(h_i, l_i)$ evaluates to true, and we write $w_i \vDash \phi_i(h,l)$.

We assume that inputs that satisfy the same path condition ϕ_i induce traces that are observationally equivalent, and hence belong to the same trace class T_i. I.e., for inputs (h,l) and (h',l'), if $(h,l) \vDash \phi_i$ and $(h',l') \vDash \phi_i$, then running $P(h,l)$ and $P(h',l')$ results in traces t_1 and t_2 that reside in the same trace class T_i. Thus, our characterization of trace classes is defined by the set of path constraints Φ, where each path constraint ϕ_i is associated with a trace class T_i. Recalling the example from Figure 6.1, the trace class characterization based on path constraints is $\{h \leq l, h > l\}$.

In the following two subsections we discuss how we address the issues of observation noise and observation collision for the trace classes induced by path constraints.

6.3.3 Estimating Observation Noise

In order to estimate the trace-observation noise, for each ϕ_i we find a witness that satisfies ϕ_i and profile the program running in the environment with that input. Witness generation is a common practice in symbolic execution in order to provide concrete inputs

Algorithm 15 System S, Path Constraints Φ, # of Samples m

```
1: procedure PROFILE(S, Φ, n)
2:     for each φ_i ∈ Φ
3:         (h_i, l_i) ← generateWitness(φ_i)
4:         for j from 1 to m
5:             sample(i, j) ← S(h, l)
```

that demonstrate a particular program behavior. Many satisfiability solvers support witness generation. Our method relies on the assumption that any witness $w_i = (h_i, l_i)$ for a path condition ϕ_i has a side-channel observation characteristic that is representative of all other inputs that satisfy ϕ_i. That is,

$$(h, l) \vDash \phi_i \Rightarrow p(O|H = h, L = l) = p(O|H = h_i, L = l_i)$$

Trace Class Sampling. Assuming that P is deterministic, every (h, l) pair results in exactly one trace, and the side-channel observation relationship for every (h, l) pair is characterized by a witness w_i for the path constraint ϕ_i that corresponds to trace class T_i. Hence, we assume that $p(O|T \in T_i) = p(O|H = h_i, L = l_i)$. Thus, in order to estimate the effect of system noise on a trace class, we repeatedly collect observation samples using w_i as a representative input (Procedure 15).

Distribution Estimation. Given a sample set of observations, $\mathcal{A}$ can estimate the probability of an as-yet-unseen observation using well-known density function interpolation methods using *smooth kernel density estimation*. Suppose $\{o_1, o_2, \ldots, o_n\}$ is a set of independent and identically distributed samples drawn from the unknown distribution $p(O|T \in T_i)$ for a specific trace class T_i. We want to estimate the value of $p(O = o|T \in T_i)$ for an unseen sample o. The *kernel density estimator* $\hat{p}(O|T \in T_i)$ is

$$\hat{p}(O = o|T \in T_i) = \frac{1}{nh} \sum_{i=1}^{n} K\Big(\frac{o - o_i}{h}\Big) \tag{6.5}$$

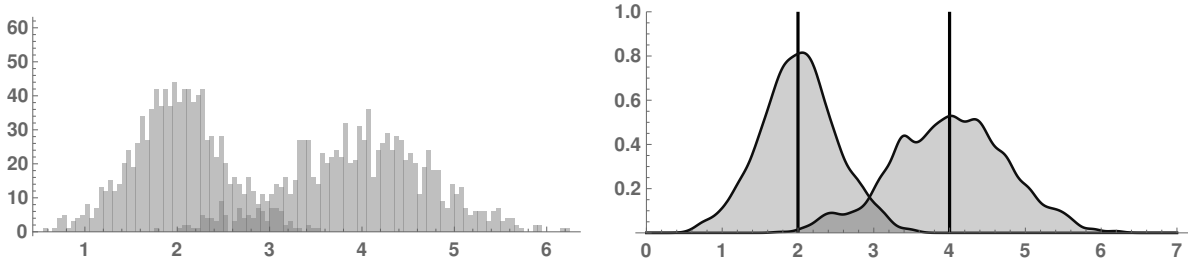


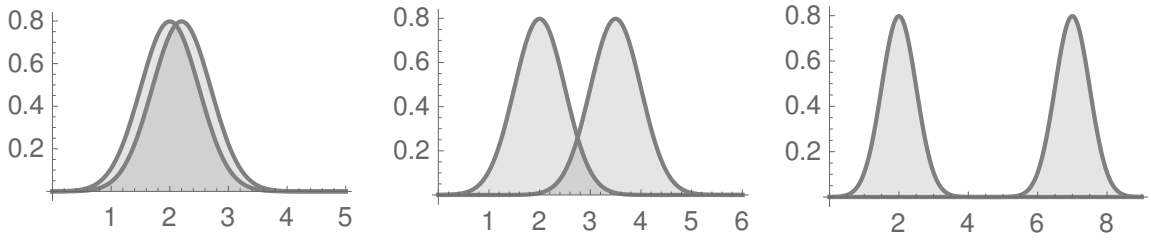
Figure 6.6: Histogram of 1000 timing samples for trace classes T_1 and T_2 (left) and the smooth kernel density estimates for $p(O|T \in T_i)$ (right).

where K is a smoothing kernel and h is a smoothing parameter called the bandwidth [95, 96]. We use a Gaussian distribution for the smoothing kernel K and we use a bandwidth that is inversely proportional to the sample size. Using Procedure 15 and Equation 6.5, we have an estimate for the effect of the noise on each trace class T_i.

For the example from the introduction, there are two trace classes T_1 and T_2 corresponding to path conditions $\phi_1(h, l) = h \leq l$ and $\phi_2(h, l) = h > l$. We use a constraint solver to find witnesses $w_1 = (4, 10)$ and $w_2 = (9, 3)$ so that $w_1 \models \phi_1$ and $w_1 \models \phi_2$. Then, running the system with w_1 and w_2 for 1000 timing samples results in the histograms and the the corresponding smooth kernel estimate distributions shown in Figure 6.6.

6.3.4 Trace Class Merging Heuristic

It is possible that the set of path constraints Φ that characterize $\mathcal{T}$ are an over-refinement of the actual trace classes of P. It may be that two trace classes are effectively indiscernible via observation due to system noise. For this reason, we employ a heuristic which combines path constraints when their corresponding estimated probability distributions are too similar. We measure the similarity of two distributions using the Hellinger distance $d_H(p, q)$ between density functions p and q given by



(a) $d_H = 0.068$ (b) $d_H = 0.491$ (c) $d_H = 0.978$

Figure 6.7: Three pairs of probability distributions and their Hellinger distances. With a threshold of $\tau = 0.1$, the distribution pair in (a) are indistinguishable, while distributions pairs in (b) and (c) are distinguishable.

$$d_H(p,q) = \sqrt{\frac{1}{2}\int_{-\infty}^{\infty}\left(\sqrt{p(x)} - \sqrt{q(x)}\right)^2 dx}$$

The Hellinger distance is such that $0 \leq d_H(p_1, p_2) \leq 1$. Intuitively, $d_H(p_1, p_2)$ measures distance by the amount of "overlap" between p and q: $d_H(p,q)$ is 0 if there is perfect overlap and 1 if the two distributions are completely disjoint (see Figure 6.7). We merge path conditions ϕ_i and ϕ_j if $d_H(\hat{p}(O|T \in T_i), \hat{p}(O|T \in T_j)) \leq \tau$ for a threshold τ.

6.4 Online Attack Synthesis

In this section we describe how to automatically synthesize an attack for the adversary.

6.4.1 Adversary Strategy

Recall the overall system model from Figure 6.4 in which the adversary $\mathcal{A}$ repeatedly interacts with the system, trying to discover information about h. We define the high level strategy for the adversary $\mathcal{A}$ in Algorithm 16 and summarize the steps taken in our model of $\mathcal{A}$.

1. **Input choice.** $\mathcal{A}$ executes an attack step by choosing an input l^* that will maximize the expected information leakage. The method for choosing l^* is the core of

Algorithm 16 System S, Current belief $p(H)$

```
1: procedure ATTACK(S, p(H))
2:     l ← chooseLowInput(p(H))
3:     o ← S(h, l)
4:     p(H) ← p(H|O = o, L = l)
5:     ATTACK(S, p(H))
```

our technique, involving the numeric maximization of an objective leakage function which is symbolic over l, computed via symbolic weighted model counting. Sections 6.4.2, 6.4.3, and 6.4.4 detail this step.

2. **Program invocation.** The adversary submits input l so that $P(h, l)$ is invoked.

3. **Record observation.** As a result of invoking the system, $\mathcal{A}$ records a side-channel observation o.

4. **Belief update**. $\mathcal{A}$ uses the current belief $p(H)$ and trace class probabilities to make a Bayesian update on the current belief about h given input l^* and observation o. That is, $\mathcal{A}$ sets the current belief $p(H = h) \leftarrow p(H = h|O = o, L = l)$ (Section 6.4.5).

5. **Repeat.** Run the attack on S with the updated belief.

The step $l \leftarrow \text{chooseLowInput}(p(H))$ is an entropy maximization procedure consisting of several components. We use weighted model counting to generate expressions that quantify the probabilities of trace classes given an input l, which is kept symbolic, and the current belief about the secret, $p(H)$. These probability expressions can be used along with the estimated noise distributions to express the mutual information $\mathcal{I}(O; H|L = l)$. Alternatively, as a heuristic, one may consider the mutual information with regard to trace classes, $\mathcal{I}(T; H|L = l)$. In either case, we use numeric maximization routines to find the input which maximizes $\mathcal{I}$. Our experiments demonstrate that maximizing

$\mathcal{I}(T; H|L = l)$ is more scalable since it is considerably faster and still generates efficient attacks.

6.4.2 Trace Class Probabilities via Symbolic Weighted Model Counting

When $\mathcal{A}$ selects an input l, the system runs $P(h, l)$ resulting in a trace which cannot be directly known to $\mathcal{A}$ using side-channel observations. However, $\mathcal{A}$ can compute the probability that the trace belongs to trace class T_i, i.e., $p(T \in T_i|L = l)$, relative to his current belief about the distribution of the secret $p(H)$. If P is deterministic, for a concrete choice of l the probability of a particular trace t relative to the current belief about h is simply the probability of the particular value of h that induces t; $p(T = t_i|L = l) = p(H = h)$. Furthermore, recall from Section 6.3.2 that each trace class T_i is associated with a path constraint on the inputs $\phi_i(h, l)$, and think of $\phi_i(h, l)$ as a characteristic function that returns 1 if running $P(h, l)$ results in a trace t from trace class T_i and returns 0 otherwise. Then, we have the following equivalence between trace class probabilities with respect to a choice of l, the current belief $p(H)$ and the path constraints Φ:

$$p(T \in T_i|L = l) = \sum_{t \in T_i} p(T = t_i|L = l) = \sum_{h \in \mathbb{H}} p(H = h)\phi_i(h, l) \tag{6.6}$$

The resulting expression on the right hand side is an instance of a *weighted model counting* problem. Unweighted model counting is the problem of counting the number of models of a formula. If one assigns a weight to each model, one may compute the sum of the weights of all models. Thus, the unweighted model counting problem is a special instance of the weighted model counting problem obtained by setting all weights equal to 1. In our case, we are interested in counting models of ϕ_i where the weight of each model (h, l) is given by $p(H = h)$.

Clearly, one may compute Equation (6.6) by summing over $\mathbb{H}$ but this would be inefficient. For instance, $\mathbb{H}$ may be large and Equation (6.6) will need to be recomputed every time $p(H)$ is updated. In addition, in the numeric maximization procedure, which we describe later, we would like to be able to evaluate Equation (6.6) for many different values of l. Hence, we seek an efficient way to compute the weighted model count. This is accomplished by using symbolic weighted model counting. We first give an example.

Recall the path constraints from the example in Figure 6.1 where $\phi_1(h, l) = h \leq l$ and $\phi_2(h, l) = h > l$ corresponding to two trace classes T_1 and T_2. Additionally, suppose that $\mathcal{A}$'s current belief is that $1 \leq h \leq 24$ and that larger values of h are more likely to occur. $\mathcal{A}$ models his initial belief as a probability distribution

$$p(H = h) = \begin{cases} \frac{h}{300} & 1 \leq h \leq 24 \\ 0 & \text{otherwise} \end{cases}$$

We can compute symbolic formulas for $p(T \in T_i | L = l)$ as functions of $\mathcal{A}$'s choice of input l:

$$p(T \in T_1 | L = l) = \begin{cases} 0 & l < 1 \\ \frac{l^2+l}{600} & 1 \leq l \leq 24 \\ 1 & l > 24 \end{cases}, \qquad p(T \in T_2 | L = l) = \begin{cases} 1 & l < 1 \\ 1 - \frac{l^2+l}{600} & 1 \leq l \leq 24 \\ 0 & l > 24 \end{cases}$$

When $\mathcal{A}$ wants to compute the probability that the program will execute a trace from any trace class for a particular choice of l, he may simply evaluate these functions for that choice of l. For example, if $\mathcal{A}$ inputs $l = 10$ he expects to observe a trace from T_1 with probability 11/60 and a trace from T_2 with probability 49/60. Using trace class probability functions that depend on l, $\mathcal{A}$ can efficiently compare the likelihood

that the program executes a trace from any trace class depending on the choice of l. These symbolic probability functions are used in the next section to generate a symbolic information gain function which is used as the objective of numeric maximization.

We compute a symbolic function for $p(T \in T_i | L = l)$ by using a model-counting constraint solver. Our implementation targets functions whose path constraints can be represented as boolean combinations of linear integer arithmetic constraints and we use the model counting tool Barvinok [64, 58] for this purpose. For programs that operate on strings, we interpret strings as arrays of integers that are bounded to be valid byte values of ASCII characters.

Barvinok performs weighted model counting by representing a linear integer arithmetic constraint ϕ on variables $X = \{x_1, \ldots, x_n\}$ with weight function $W(X)$ as a symbolic polytope $Q \subseteq \mathbb{R}^n$. Let $Y \subseteq X$ be a set of parameterization variables and Y' be the remaining free variables of X. Barvinok's polynomial-time algorithm generates a (multivariate) piecewise polynomial F such that $F(Y)$ evaluates to the weighted count of the assignments of integer values to Y' that lie in the interior of Q. We are interested in computing the probability of a trace class given a choice of l and the current belief about the high security inputs. Thus, we let $Y = \mathbb{L}$, $Y' = \mathbb{H}$, and W be $p(H = h)$.

6.4.3 Leakage Objective Function

Here we describe how to generate an objective function that quantifies the amount of information that $\mathcal{A}$ can gain by observing o after sending input l. The major point of this section is to show that it is possible to express the information leakage as a symbolic function using weighted model counting functions and the estimated noise distributions.

Mutual Information Between Secret and Observation. For a given choice of l we can quantify $\mathcal{I}(H; O | L = l)$ by directly applying the definition of mutual information,

Equation (6.4).

$$\sum_{h \in \mathbb{H}} p(H = h) D_{\mathrm{KL}}(p(O = o | H = h, L = l), p(O = o | L = l))$$

Because the path constraints Φ are disjoint, and cover all paths of the program, for a particular l, Φ determines a partition on h. Thus, we can rewrite $\mathcal{I}(H; O | L = l)$ by summing over path conditions:

$$\sum_{i=1}^{n} \sum_{h \in \mathbb{H}} p(H = h) \phi_i(h, l) D_{\mathrm{KL}}(p(O = o | H = h, L = l), p(O = o | L = l))$$

Since $\phi_i(h, l) = 0$ unless input (h, l) induces trace class T_i and the observation probability is conditioned on the trace class, we rewrite the expression in terms of trace classes:

$$\sum_{i=1}^{n} D_{\mathrm{KL}}(p(O = o | T \in T_i), p(O = o | L = l)) \sum_{h \in \mathbb{H}} p(H = h) \phi_i(h, l)$$

Observe that the sum over $\mathbb{H}$ is exactly that which is computed via symbolic weighted model counting in Equation (6.6); thus, $\mathcal{I}(H; O | L = l)$ can be expressed as

$$\sum_{i=1}^{n} D_{\mathrm{KL}}(p(O = o | T \in T_i), p(O = o | L = l)) p(T \in T_i | L = l) \tag{6.7}$$

where D_{KL} is computed with Equation (6.3) and the probability of the observation conditioned on the low input choice is a straightforward conditional probability computation

$$p(O = o | L = l) = \sum_{i=1}^{n} p(O = o | T \in T_i) \cdot p(T \in T_i | L = l) \tag{6.8}$$

Thus, $\mathcal{I}(H; O | L = l)$ can be computed using Equations (6.7) and (6.8) using only $p(O = o | T \in T_i)$ (estimated via sampling and smooth kernel interpolation) and $p(T \in T_i | L = l)$ (computed via symbolic weighted model counting).

Mutual Information Between Secret and Trace Classes. The approach to mutual information quantification based on Equation (6.7) relies on integrating over the domain $\mathbb{O}$, an expensive computation. Alternatively, as a heuristic, one may compute the mutual information between the secret and the trace classes given low input.

$$\mathcal{I}(H;T|L=l) = \mathcal{H}(T|L=l) - \mathcal{H}(T|H,L=l)$$

Note that if P is deterministic, T is completely determined by H and L and so $\mathcal{H}(T|H,L=l)=0$. Thus,

$$\mathcal{I}(H;T|L{=}l) = \sum_{i=1}^{n} p(T{\in}T_i|L{=}l)\log\frac{1}{p(T{\in}T_i|L{=}l)} \tag{6.9}$$

While this is not guaranteed to give the optimal attack, it can by computed much more efficiently than Equation (6.7). Equation (6.9) can be quickly evaluated for many choices of l using the symbolic functions for $p(T{\in}T_i|L{=}l)$ because they are computed by Barvinok as symbolic weighted model counting functions. In our experiments we will see that quantifying information leakage in this way allows for more efficient generation of attack inputs.

Recall the example from Figure 6.1. After using symbolic execution to determine path constraints associated with trace classes, estimating the noise, and performing symbolic weighted model counting, we can compute both $\mathcal{I}(H;O|L=l)$ and $\mathcal{I}(H;T|L=l)$, plotted in Figure 6.8. Observe that the expected information leakage computed using Equation (6.7) is strictly less than the one computed using only trace classes based on Equation (6.9). So, the trace class information gain bounds the actual information gain. However, both maxima occur at the same value of $l^* = 17$. Hence, both methods agree on the optimal choice of input. While we do not claim that the two maxima

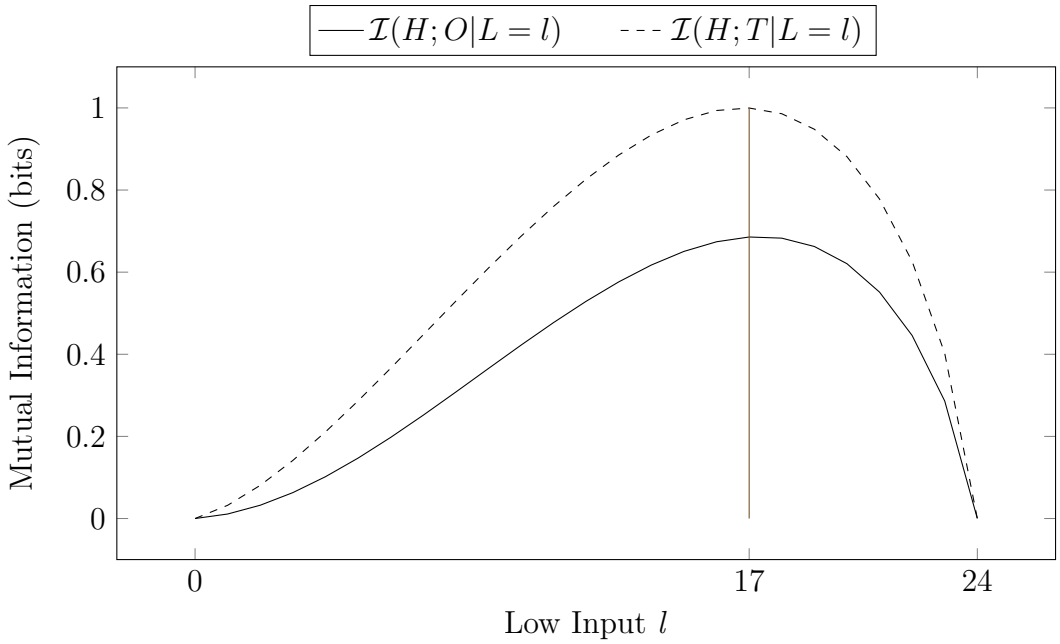


Figure 6.8: Mutual information between secret H and observation O or trace classes T as a function of low input $L = l$ for the example from Figure 6.1.

Algorithm 17 Noise-Entropy Aware Input Choice. Current belief $p(H)$, path constraints Φ, noise $p(O|L)$

```
1: procedure CHOOSELOWINPUT1(p(H), Φ, p(O = o|L = l))
2:     for each φ_i ∈ Φ
3:         p(T ∈ T_i|L = l) ← BARVINOK(p(H), Φ)
4:     f(l) ← I(H; O|L = l) via Eq. 6.7
5:     l* ← NMAXIMIZE(f(l))
6:     return l*
```

coincide in general, our experiments support that maximizing $\mathcal{I}(H;T|L=l)$ rather than $\mathcal{I}(H;O|L=l)$ is significantly faster per attack step, but may result in slightly longer attack sequences.

6.4.4 Input Choice via Numeric Optimization

Now, using Equation (6.7) or Equation (6.9), $\mathcal{A}$ can apply numeric optimization procedures to choose an input l^* that maximizes the information gain. $\mathcal{A}$ can compute

Algorithm 18 Noise Entropy Agnostic Input Choice. Current belief $p(H)$, path constraints Φ

```
1: procedure CHOOSELOWINPUT2(p(H), Φ)
2:     for each φ_i ∈ Φ
3:         p(T ∈ T_i | L = l) ← BARVINOK(p(H), Φ)
4:     f(l) ← I(H; T | L = l) via Eq. 6.9
5:     l* ← NMAXIMIZE(f(l))
6:     return l*
```

$$l^* = \arg\max_l \mathcal{I}(H; T|L = l) \quad \text{or} \quad l^* = \arg\max_l \mathcal{I}(H; O|L = l)$$

These are non-linear combinatorial optimization problems. We can use black-box objective function maximization routines to find the values of l that maximize the leakage. These routines are typically stochastic, and guaranteed to produce a local optimum, but not necessarily a global one. In our implementation, we use MATHEMATICA's NMAXIMIZE function, further described in our implementation section. Regardless of how l^* is chosen, $\mathcal{A}$ can still efficiently and precisely update $p(H)$ based on l^* and the resulting system side-channel observation which we detail in Section 6.4.5. The two methods given as Procedures 17 and 18.

6.4.5 Belief Update for Secret Distribution

After providing input l^* and making side-channel observation o, $\mathcal{A}$ will need to refresh his current belief about the secret $p(H=h)$ by performing the update $p(H=h) \leftarrow p(H=h|O=o, L=l)$. Although this is a straightforward Bayesian update, we provide the formula in order to illustrate how $p(T \in T_i|L=l)$, computed via weighted model counting, and $p(O=o|T \in T_i)$, estimated via Procedure 15, are involved in the update. By applying Bayes' rule and the

definitions of conditional probability and summing over trace classes we have that

$$p(H{=}h|O{=}o, L{=}l^*) = \sum_{i=1}^{n} p(H{=}h)\phi_i(h,l)\frac{p(O{=}o|T \in T_i)}{p(O{=}o|L{=}l^*)} \tag{6.10}$$

where $p(O = o|L{=}l)$ is computed from $p(T \in T_i|L = l)$ and $p(O{=}o|L{=}l)$ using Equation (6.8). Equation (6.10) allows $\mathcal{A}$ to easily update $p(H)$ by simply plugging the value of l^* that was used as input and the resulting observation o.

6.4.6 Example

Recall the example from Figure 6.1 and suppose $\mathcal{A}$ has an initial belief about the secret: $p(H = h) = h/300$ if $1 \leq h \leq 24$ and 0 otherwise (Figure 6.9, far left). There are two path constraints $\phi_1(h,l) = h \leq l$ and $\phi_2(h,l) = h > l$, which result in trace class probability functions $p(T \in T_i|L = l)$ as in Section 6.4.2.

As we saw in Section 6.4.3, the optimal input of the first step of an attack is $l_1^* = 17$. In Figure 6.9 we show 6 steps of an attack in which the secret is $h = 20$ and we start by making input $l^* = 17$. Suppose that when making the input, $\mathcal{A}$ makes a timing measurement and observes $o_1 = 3.1$ms Then, $\mathcal{A}$ updates his belief about h as shown in the second step of Figure 6.9. Recalling the noise estimates in Figure 6.6, we see that for a timing measurement of 3.1ms, it is more likely that a trace from trace class T_2 corresponding to $\phi_2(h,l) = h > 17$ occurred, than a trace from trace class T_1 corresponding to $\phi_1(h,l) = h \leq 17$. Consequently, after performing the Bayesian update, the probability mass to the right of $h = 17$ increases and the probability mass to the left decreases proportionally according to Equation (6.10).

$\mathcal{A}$ continues choosing inputs and making observations for 6 steps. We see that at different steps of the attack, the optimal input is repeated from a previous step: $l_4 = l_6 = 19$, for instance; this technique automatically performs a form of repeated sampling

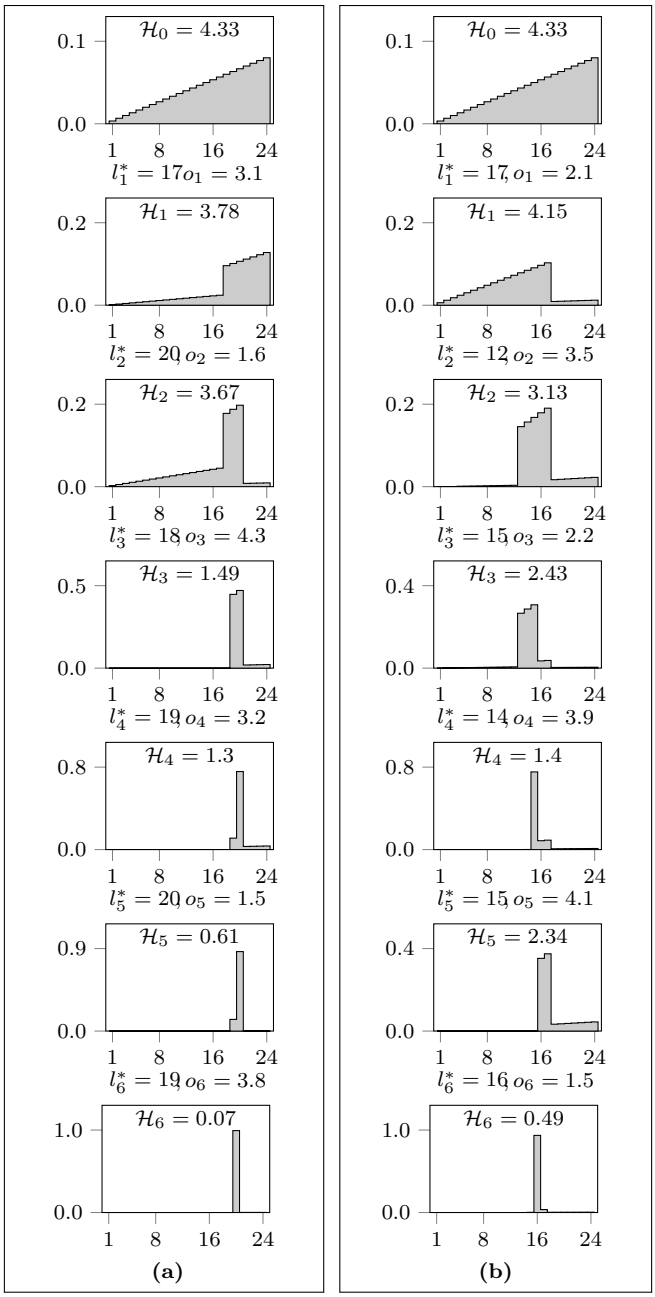


Figure 6.9: Two sequences of attack steps, indicating $\mathcal{A}$'s changing belief about the secret $p(H = h)$ after making input l_i^* and observing o_i at the i^{th} step. The current uncertainty, $\mathcal{H}_i$ (bits), is indicated.

in order to improve confidence. After 6 steps the probability mass of $p(H = h)$ is concentrated on $h = 20$ and the corresponding uncertainty is $\mathcal{H}_6 = 0.07$ bits, and so $\mathcal{A}$ may reasonably conclude that $h = 20$ with high probability.

As a final comment on this example, we note that chooseLowInput1, which optimizes based on noise, and chooseLowInput2, which optimizes based only on trace classes, both give identical attack sequences if provided the same observation sequences from Figure 6.9.

6.4.7 Handling Non-deterministic Programs

Although our model assumes a deterministic program, we are able to handle non-deterministic programs in some cases. Programs may contain explicit randomization through the use of random number generators. Indeed, there are attempts to mitigate side-channel leakage by introducing randomization into the program [97, 98]. If a program has a random component that does not affect the branch conditions on h and l, then the randomization is essentially decoupled from the computation on h and l. Thus, running symbolic execution on such a program will yield path constraints on h and l that can be used to characterize trace classes. Then, dynamic profiling with witnesses for each path constraint will capture the effect of non-deterministic choices in the code on the observation. Hence, the deterministic component of the program that computes over h and l and the random component can be factored into the static analysis of symbolic execution and the dynamic analysis of profiling, respectively. We demonstrate in our experiments that we are indeed able to use our analysis on programs which contain explicit non-determinism in the code.

6.4.8 Detecting Non-vulnerability

Some programs are not vulnerable to adaptive side-channel attacks with respect to a chosen observable. We demonstrate our ability to report non-vulnerability in our experimental evaluation. Our approach is able to report non-vulnerability in two ways:

1. If all trace classes initially defined by the path constraints are determined to be observationally indistinguishable using the Hellinger distance merging metric, we conclude that there is only one trace class, and therefore, different inputs cannot leak information about the secret.

2. If the first step of attack generation determines that there is no input l^* which results in positive information gain, then again we conclude that there is no attack.

Note that trace-class constraints are generated with respect to a semantic model of a program which is an abstraction of real system behavior. In our implementation, our semantic model is based on path constraints generated from branch instructions in Java byte-code, and so our abstraction does not capture lower level details like thread scheduling or caching. The power of our technique is relative to the abstraction level used in constraint generation and the cost model. Hence, our reports of non-vulnerability are made with respect to the chosen level of semantic abstraction.

6.5 Implementation and Experimental Setup

I implemented this technique according to the high-level diagram shown in Figure 6.5 and described in Section 6.2.2. We ran our attack synthesis system on client-server programs created by DARPA (Defense Advanced Research Projects Agency) for the ongoing STAC (Space/Time Analysis for Cybersecurity) [99] research program. We evaluated

the effectiveness of our approach on several programs taken from their publicly available repository [2]. These programs were crafted to test and evaluate techniques for detecting side-channel vulnerabilities in networked Java applications.

Reference platform. For all experiments, the system under test (SUT) was run on the official DARPA STAC Reference Platform [2], which specifies an Intel NUC 5i5RYH computer with an Intel Core i5-5250 CPU running at 1.60 GHz, 16 GB of DDR3 RAM, and its built-in Intel 1000 Mbps Ethernet interface. The reference operating system is CentOS 7, release 7.1.1503, with a Linux 3.10.0-229 64-bit kernel. We used the OpenJDK 64-bit Java VM, build 1.8.0_121.

Trace class extraction. We infer trace classes by symbolically executing the SUT source code. We used Symbolic Path Finder (SPF), an extension to NASA's Java Path Finder v8.51 using the Z3 SMT-solver [29], version 4.4.1.

Automated profiling. This involves two components: a client-side Profiler and a server-side AppServer. The server component is a wrapper for DARPA STAC canonical challenge programs. Although their interfaces were slightly modified to achieve a homogeneous input/output format, the core implementation of each challenge program was left unmodified. The client-side Profiler is a Python script that invokes the server while taking measurements. Given a list of trace-class witnesses, and the number of samples per witness, the Profiler configures the server and repeatedly interacts with it over the network, carefully timing each interaction. For our experiments, each trace class witness is used to sample the system 1000 times.

Model counting. Symbolic weighted model-counting functions are computed by sending the path constraints and $p(H{=}h)$ to the Barvinok model counter [58], version v0.39.

Numeric and symbolic computing. Many computationally intensive numeric and symbolic operations are handled by Mathematica, including smooth kernel probability

density estimation from timing samples, symbolic manipulation of model-counting and information-theoretic functions, numeric integration, objective function maximization, and Bayesian updating. All experiments were run using Wolfram Mathematica v11 [100] on a Linux 64-bit system.

Entropy maximization. We used Mathematica's NMaximize function [94] in *Differential Evolution* mode, which uses a fast and robust genetic algorithm [89, 101].

6.6 Experiments

6.6.1 DARPA-STAC Benchmark

Our experiments show that our approach is able to dynamically synthesize attack input sequences for vulnerable programs and to report non-vulnerability for programs for which an attack is not feasible. In Table 6.1, we label the DARPA benchmark programs with a number that is consistent with the numbering from their repository along with '(v)' or '(nv)' to indicate vulnerable or non-vulnerable programs according to DARPA's classification. $|\Phi|$ denotes the number of path conditions, $|\mathcal{T}|$ denotes the number of trace classes after performing Hellinger-based merging, and $\mathrm{Dim}(h)$ indicates the size of the symbolic integer vector used to represent h. For STAC programs 1, 3, and 11, we varied the secret search domain $(2^8, 2^{16}, 2^{24}, 2^{31})$, while keeping the code the same, and so only a single offline phase was needed. For instance, row 1 of Table 6.1 corresponds to the offline phase of 4 different online attacks.

The STAC benchmark also contains programs that are not related to side-channel problems, so we did not analyze them. We analyzed only problems marked by DARPA as side-channel related and that fit our model. There were two side-channel problems that require a different cost observation model than ours, so we did not analyze those

Table 6.1: Experimental data for publicly available STAC benchmarks [2].

							Offline Phase Time (seconds)			
Benchmark		Dim(H)	$\lvert\mathbb{H}\rvert$	$\lvert\Phi\rvert$	$\lvert\mathcal{T}\rvert$	Vuln?	S.E.	Noise Est.	Merging	Total
1	STAC-1(nv)	1	$2^{\{8,16,24,31\}}$	2	1	no	0.57	22.28	0.81	23.67
2	STAC-3(nv)	1	$2^{\{8,16,24,31\}}$	6	3	no	0.64	36.18	4.89	41.72
3	STAC-1(v)	1	$2^{\{8,16,24,31\}}$	2	2	yes	0.56	31.52	0.48	32.58
4	STAC-3(v)	1	$2^{\{8,16,24,31\}}$	6	4	yes	0.57	34.09	5.17	39.85
5	STAC-11A(v)	1	$2^{\{8,16,24,31\}}$	3	2	yes	0.58	25.65	1.32	27.56
6	STAC-11B(v)	1	$2^{\{8,16,24,31\}}$	3	2	yes	0.57	26.63	1.29	28.50
7	STAC-4(v)	1	26	10	2	yes	0.73	14.79	7.10	22.63
8	STAC-4(v)	2	702	27	3	yes	1.19	44.52	2.28	48.01
9	STAC-4(v)	3	18278	55	5	yes	2.67	100.55	64.94	168.17
10	STAC-12(v)	1	26	17	4	yes	0.94	26.30	18.57	45.83
11	STAC-12(v)	2	702	39	5	yes	0.99	57.46	48.67	107.13
12	STAC-12(v)	3	18278	77	6	yes	1.62	125.49	132.63	259.76
13	STAC-12(v)	4	475254	149	7	yes	3.06	258.48	293.57	555.13

programs. Initial secret distributions in our experiments are uniform.

Non-vulnerable programs. In Table 6.1 we present results of running our attack synthesizer on two non vulnerable programs. We see that STAC–1(nv) (source code in Figure 6.10) is not vulnerable to an adaptive timing side-channel attack because there is only one trace class and so there are no attack steps taken. On the other hand, our tool tells us that STAC–3(nv) (source code in Figure 6.11) is not vulnerable, despite having 3 observationally distinguishable trace classes, because after 1 attack step there are no inputs which can leak any information. Thus, we agree with the DARPA non-vulnerable classification. For both programs we observe that the majority of the analysis time is spent in offline profiling.

Optimizing observation vs trace-class entropy. We applied both ChooseLowInput1, which optimizes based on observation entropy, and ChooseLowInput2, which optimizes based on trace class entropy, for two STAC programs. STAC–1(v) is similar to our running example with a branch condition $h \leq l$ in which extra computation is done.

STAC–3(v) is a program that contains 6 different branches and an internal parameter n which, depending on h and l, causes 2 branches to run in $O(1)$ time, 1 branch to run in $O(n)$ time, 1 branch to run in $O(n^2)$ time, and 2 branches to run in $O(n^3)$ time.

```
public static String category1_nv
 (int[] secret, int[] guess) {
  int n = 3;
  if(guess[0] <= secret[0]){
    for(int i=0;i<n;i++){
      for(int t=0;t<n;t++) {
        sleep(1);
      }
    }
  }
  else{
    for(int i=0;i<n;i++){
      for(int t=0;t<n;t++) {
        sleep(1);
      }
    }
  }
  return "DONE";
}
```

```
public static String category1_v
 (int[] secret, int[] guess) {
  int n = 3;
  if(guess[0] <= secret[0]){
    for(int i=0;i<n;i++){
      for(int t=0;t<n;t++) {
        sleep(1);
      }
    }
  }
  else{
    for(int i=0;i<n;i++){
      for(int t=0;t<n;t++) {
        sleep(2);
      }
    }
  }
  return "DONE";
}
```

Figure 6.10: Non-vulnerable (left) and vulnerable (right) versions of STAC–1.

```
public static String category3_nv
 (int[] secret, int[] guess_t) {
  int n = 2;
  int guess  = guess_t[0];
  int t = guess_t[1];
  if(guess <= secret[0]){
    if(t == 1){sleep(1);}
    else if(t == 2){
      for(int i = 0; i<n*n;i++){
        sleep(1);
      }
    }
    else{
      for(int i = 0; i<n*n*n;i++){
        sleep(1);
      }
    }
  }
  else{
    if(t == 1){sleep(1);}
    else if(t == 2){
      for(int i = 0; i<n*n;i++){
        sleep(1);
      }
    }
    else{
      for(int i = 0; i<n*n*n;i++){
        sleep(1);
      }
    }
  }
  return "DONE";
}
```

```
public static String category3_v
 (int[] secret, int[] guess_t) {
  int n = 2;
  int guess  = guess_t[0];
  int t = guess_t[1];
  if(guess <= secret[0]){
    if(t == 1){sleep(1);}
    else if(t == 2){
      for(int i = 0; i<n;i++){
        sleep(1);
      }
    }
    else{
      for(int i = 0; i<n*n*n;i++){
        sleep(1);
      }
    }
  }
  else{
    if(t == 1){sleep(1);}
    else if(t == 2){
      for(int i = 0; i<n*n;i++){
        sleep(1);
      }
    }
    else{
      for(int i = 0; i<n*n*n;i++){
        sleep(1);
      }
    }
  }
  return "DONE";
}
```

Figure 6.11: Non-vulnerable (left) and vulnerable (right) versions of STAC–3.

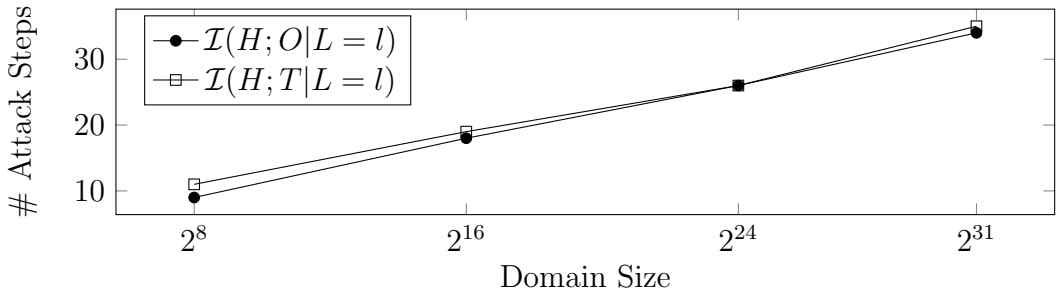


Figure 6.12: STAC–1(v) attack steps: observation vs. trace class entropy.

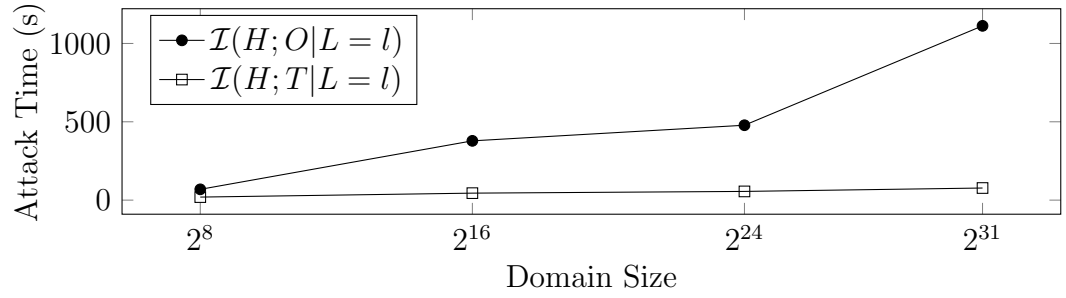


Figure 6.13: STAC–1(v) attack time: observation vs. trace class entropy.

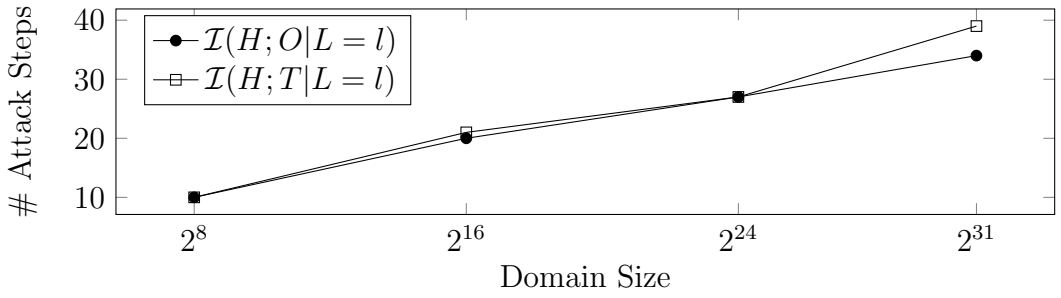


Figure 6.14: STAC–3(v) attack steps: observation vs. trace class entropy.

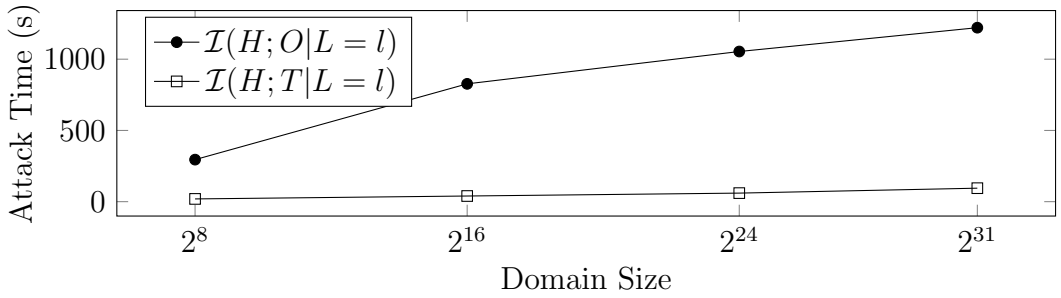


Figure 6.15: STAC–3(v) attack time: observation vs. trace class entropy.

The low input to the program consists of 2 values. The offline phase merged the path constraints for branches with the same time complexities into the same trace classes. The online attack phase automatically chose inputs which caused the program to always either execute the $O(n)$ or $O(n^2)$ branch in order to leverage the timing difference and fully leak the secret.

The offline phase data are shown in Table 6.1 and online attack phase data in Figures 6.13 and 6.15. The online attack phase was run over different secret domain sizes. For each domain size we ran until $p(H = h)$ converged with certainty to a single value, which we manually verified was the correct secret. We observe that in all cases, optimizing for trace class entropy generates an attack that takes either the same number of steps or is slightly longer. However, optimizing for trace class entropy synthesizes attack steps much more quickly. For both STAC–1(v) and STAC–3(v), we were able to synthesize attacks for domains of size 2^{31} in under 2 minutes, including offline and online phases. Because optimizing trace class entropy is significantly faster and generates strong attacks, the remaining experiments were run only with ChooseLowInput2.

Attack synthesis with programmatic non-determinism. Program STAC–11(v) has 2 versions, A and B. Both versions contain explicit recursive randomization that affects the running time of the application (source code is Figures 6.16, 6.17, and 6.18. In both cases, SPF is able to extract path conditions which depend only on h and l and do not depend on any randomly generated variables. Thus, as discussed in 6.4.7, the effect of the randomization on the running time is independent of the path conditions on h and l and can be determined using profiling during the offline phase which is presented in Table 6.1. In Figure 6.22 we see the running time and number of attack steps required to completely leak the secret for both versions of STAC–11(v).

Programs with segment oracle side channels. STAC–4(v) and STAC–12(v) are

```
1 public static Random r = new Random(System.currentTimeMillis());
2 private static boolean[] t = new boolean[100];
3
4 private static void randomizeT(){
5     for(int x=0; x<t.length; x++){
6         t[x] = r.nextBoolean();
7     }
8 }
```

Figure 6.16: Common code for STAC–11A(v) and STAC–11B(v).

programs with segment oracle side channel [65]. Segment oracle side channels allow an attacker to use timing measurements to incrementally leak information about contiguous segments of arrays or strings. The relevant source code for STAC–12(v) is shown in Figure 6.20. The function under test is `verifyCredentials`, which, using the function `checkChar`, is a (somewhat obfuscated) way of comparing a candidate password to an actual secret password as part of a log-in system where valid strings consist of lowercase letters. DARPA challenge problems sometimes contain `delay` functions which mimic additional computational work. At a high level, this function compares individual characters of the secret and candidate password one at a time in a loop from left to right, and the running time of the function is proportional to the number of characters which match.

For example, if the password is 'ciqa', the running time will be slightly longer if an attacker inputs 'ciqg' (first 3 characters match) vs. 'ciao' (first 2 characters match). Thus, an attacker can use a timing side channel to reveal prefixes of the secret, and reduce the $O(k^n)$ brute-force search space to $O(k \cdot n)$. Segment oracle attacks were responsible for several real-world vulnerabilities [6, 5, 79].

Our method automatically synthesizes segment oracle attacks. We ran our attack synthesis method on STAC–4(v) (source code in FIgure 6.19 and STAC–12(v) (source code in Figure 6.20 where the secret domain is strings of lowercase letters. We set an online phase timeout of 20 minutes and ran attack synthesis for increasing string lengths.

```
1  private static boolean checkSecret11A(int guess, int secret, boolean[] t)
2   throws InterruptedException {
3      recur11A(guess, t, t.length - 1);
4      if(guess <= secret){sleep(1);}
5      return guess == secret;
6  }
7
8  private static void recur11A(int guess, boolean[] t, int index){
9      if(index == 0 && t[index]){}
10     else if(t[index]){
11       recur11A(guess, t, index -1);
12     }
13 }
14
15 public static String category11A_v(int[] H, int[] L)
16   throws InterruptedException {
17     randomizeT();
18     checkSecret11A(L[0], H[0], t);
19     return "DONE";
20 }
```

Figure 6.17: Source code of STAC-11A(v).

```
1  private static boolean checkSecret11B(int guess, int secret, boolean[] t)
2    throws InterruptedException {
3      return recur11B(guess, secret, t, t.length - 1);
4  }
5
6  private static boolean recur11B(int guess, int secret, boolean [] t, int index)
7   throws InterruptedException {
8      if(index == 0 && t[index]){
9          if(guess <= secret){sleep(1);}
10         return guess == secret;
11     }
12     else if(t[index]){return recur11B(guess, secret, t, index -1);}
13     if(guess <= secret){sleep(1);}
14     return guess == secret;
15 }
16
17 public static String category11B_v(int[] H, int[] L)
18   throws InterruptedException {
19     randomizeT();
20     checkSecret11B(L[0], H[0], t);
21     return "DONE";
22 }
```

Figure 6.18: Source code of STAC-11B(v).

```
1  private static boolean verifyCredentials_4_v(String candidate){
2      for(int x = 0; x<candidate.length();x++) {
3          if(x>=password.length()||password.charAt(x) != candidate.charAt(x)){
4              return false;
5          }
6          delay_4_v();
7      }
8      return candidate.length() == password.length();
9  }
10
11 private static void delay_4_v(){
12   for (int x = 0 ; x < 10000 ; x++) {}
13 }
```

Figure 6.19: Source code for STAC–4(v).

```
1  private static String password;
2  private static int subsequentCorrect;
3  private static int exceedPasswordLen;
4  private static void delay() {
5      for (int x=0 ; x < 75000 ; x++) {}
6  }
7  private static void checkChar(String candidate, int charNumber){
8     if(charNumber > password.length())
9         exceedPasswordLen++;
10    else if(password.charAt(charNumber - 1) ==  candidate.charAt(charNumber - 1)){
11         if(subsequentCorrect+1 == charNumber){
12             subsequentCorrect++;
13             delay();
14         }
15     }
16 }
17 private static boolean verifyCredentials(String candidate){
18    subsequentCorrect = exceedPasswordLen = 0;
19    for (int x=0; x < candidate.length(); x++){
20        checkChar(candidate,x+1);
21    }
22    return subsequentCorrect == password.length() && exceedPasswordLen == 0;
23 }
```

Figure 6.20: Source code of STAC-12(v).

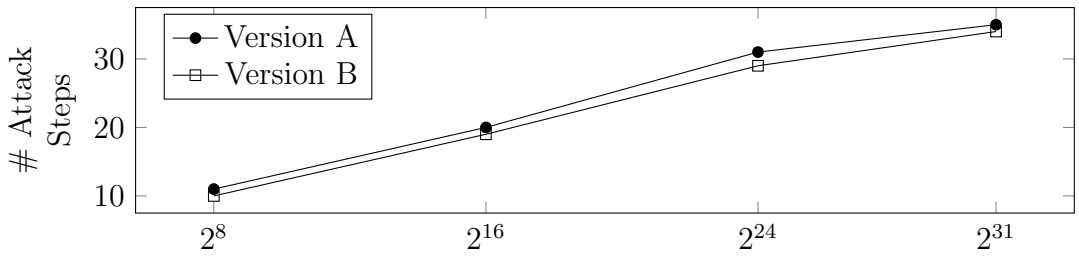


Figure 6.21: STAC–11(v) atttack steps: two versions.

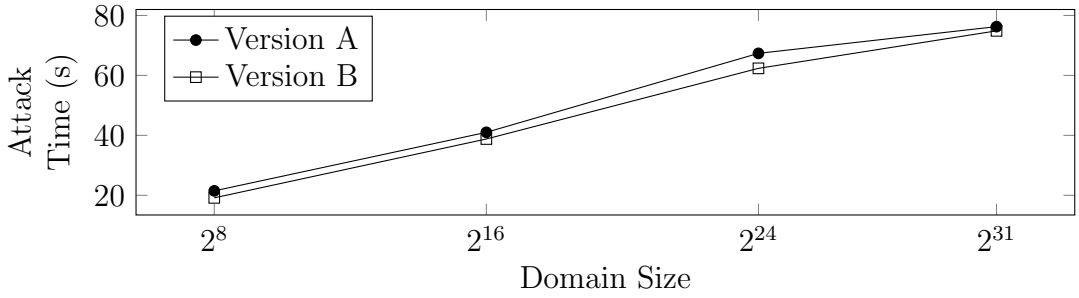


Figure 6.22: STAC–11(v) atttack time: two versions.

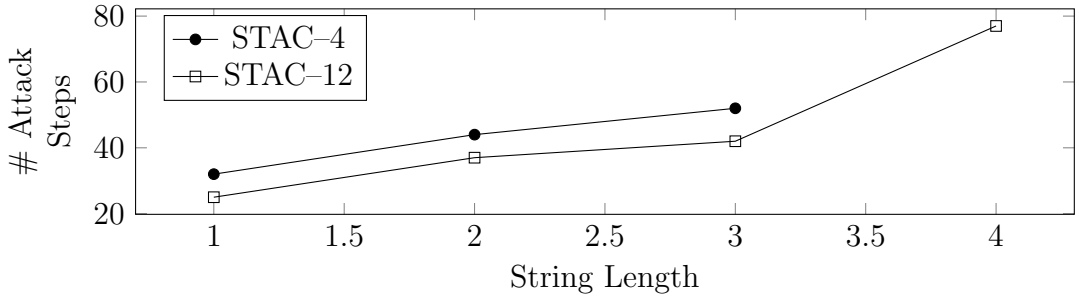


Figure 6.23: STAC–4(v) and STAC–12(v) attack steps.

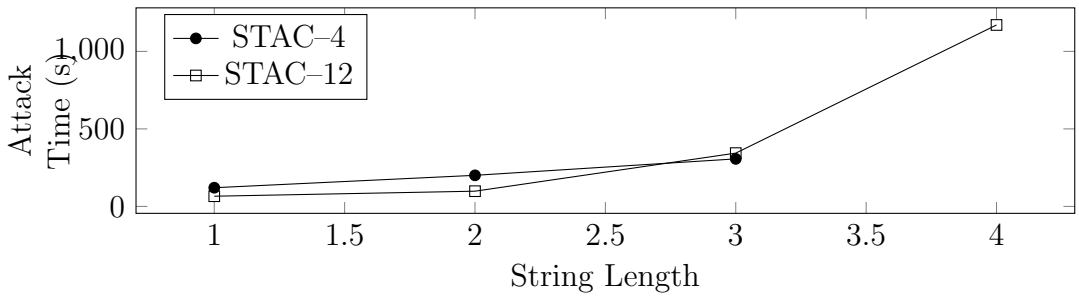


Figure 6.24: STAC–4(v) and STAC–12(v) attack time.

Table 6.2: Synthesized input strings for STAC-12(v).

Phase 1	Phase 2	Phase 3		Phase 4		Phase 5
prefix: ε	prefix: c	prefix: ci		prefix: ciq		prefix: ciqa
ε	**c**mxq	**ci**vf	**ci**sp	**ciqa**	**ciq**d	**ciqa**
daaz	**c**nte	**ci**ub	**ci**cz	**ciq**c	**ciq**o	**ciqa**
uaak	**c**tdo	**ci**az	**ci**bn	**ciq**k	**ciq**x	**ciqa**
ecjq	**c**vfo	**ci**gz	**ci**tz	**ciq**z	**ciq**z	**ciq**g
tzar	**ci**il	**ci**fl	**ci**jw	**ciq**s	**ciq**r	**ciqa**
fzgk	**c**sja	**ci**kt	**ci**ne	**ciq**i	**ciq**r	
zgap	**c**wcs	**ci**ij	**ci**le	**ciq**z	**ciq**i	
bnza	**c**ved	**ci**ok	**ciq**s	**ciq**d	**ciq**v	
zmna	**c**eyu	**ci**su	**ci**rx	**ciq**q	**ciq**u	
zmna		**ci**ld	**ciq**z	**ciq**z	**ciq**z	
maau		**ci**pa	**ci**hs	**ciq**z	**ciq**r	
vzsc		**ci**mq	**ciq**k	**ciq**e	**ciq**z	
qyas		**ci**da	**ci**eb			
asvr						

We generated attack steps for STAC–4(v) up to string length 3 before timing out. For STAC–12(v) we generated attack steps up to string length 4. The offline phase data for STAC–4(v) and STAC–12(v) are shown in Table 6.1 and online phase data is shown in Figure 6.24.

We give some details of one synthesized attack. For STAC12(v), length 4 (Table 6.1, benchmark 13) the secret was a randomly generated string, 'ciqa'. In Table 6.2 we manually separated the synthesized input strings into phases where the i^{th} phase roughly corresponds to inputs that match a secret prefix of length i. For instance, in phase 0, the attacker does not know any prefix of the secret password, first tests the system with ε, then begins testing inputs with different characters. At the end of phase 0, the attacker has discovered using side-channel measurements that the first character is 'c'. Thus, in phase 2, the attacker keeps the first character 'c' constant and tests other inputs until the side-channel observation indicates that the first 2 characters, 'ci', match. This continues similarly for phases 3 and 4, until all characters are discovered. This required 77 steps

overall to discover a secret from a search space of size 475254.

Observe that some strings were tested multiple times. For instance, the string 'ciqz' was synthesized several times over Phases 2 and 3, and the final secret 'ciqa' was tested several times during the last phase. Thus, we observe that our online attack synthesis technique appears to automatically discover the fact that repeated sampling can be used to eliminate spurious observations due to noise and reduce uncertainty. Indeed, real-world manually written attacks often employ repeated sampling in an attempt to eliminate noise. Finally, note that, once our automated attack synthesis approach generates a segment oracle attack for a program for a small secret length, it is easy to generalize such an attack to larger secrets by inspecting the generated attack.

6.6.2 Case Study: Law Enforcement Database

We applied our method to a larger application provided by DARPA. LawDB (Law Enforcement Database) is a network service that stores and manipulates a database used to store law enforcement personnel data associated with user IDs. Users can issue the command `SEARCH minID maxID` to query the database for IDs within a range. IDs are internally stored as public or restricted. Only public IDs within the search range will be shown to the user; restricted IDs are secret. Using our approach, we were able to synthesize a timing side-channel attack for this application that enables a public user to determine a restricted ID.

Symbolically executing the entire LawDB application (51 classes, 202 methods) was not feasible. By examining the source code, it was straightforward to locate the method for the `SEARCH` operation, which corresponds to case 8 of the method `channelRead0` of class `UDPServerHandler` (Figure 6.25). We noticed a possible side channel because the UDP request handler writes the message "`SEARCH ON RESTRICTED KEY OCCURRED`" to

a log file and throws a `RestrictedAccessException` depending on whether a user has entered a search query range which encompasses a restricted secret ID.

We extracted the `UDPServerHandler` class (Figure 6.25) and its closure of dependencies from the source code. Symbolic execution was not able to handle the provided log writing, so we supplied our own simplified code which writes the same message to a log file in order to mimic the original behavior of the function. We wrote a small driver (20 LOC) to interface our symbolic execution system and profiling server with the `UDPServerHandler` class. The driver initializes the database by inserting n unrestricted IDs and inserts one restricted ID, h. The driver then executes queries with range: `SEARCH` l_{minID} l_{maxID}. We symbolically execute the driver with h, l_{minID}, and l_{maxID} all symbolic and then profile the driver with the generated witnesses to conduct the offline phase. Then our online attack phase automatically synthesizes adaptive `SEARCH` range queries that eventually reveal the restricted ID.

Initial Exploratory Experiment. We initialized the database with two concrete unrestricted IDs (with values 64 and 85) and constrained allowed IDs to the range $[1, 100]$. The result of running attack synthesis on this configuration can be seen in Table 6.3 and intermediate snapshots of online attack inputs and belief updates are shown in Figure 6.26. We see that there are 42 path constraints generated by symbolic execution which reduce to 3 trace classes after profiling and merging. The offline phase takes less than 1 minute and the online attack phase of 25 steps takes less than 3 minutes.The secret ID was set to be a randomly generated number, 92.

Figure 6.26 illustrates interesting self-correcting behavior of the automated search. Due to observation noise, in step 5 a timing observation was made that caused the belief distribution to become highly concentrated in the range $[64, 76]$. The subsequently synthesized inputs are queries which search this interval and eventually eliminate that concentration of probability mass. Steps 12 through 25 generate queries that concentrate

the belief on the secret ID, 92.

Table 6.3: Experimental data for 4 different instantiations of the LawDB case study.

Benchmark		# IDs	$\|\mathbb{H}\|$	$\|\Phi\|$	$\|\mathcal{T}\|$	Offline Phase Time(s)				Online Phase	
						S.E.	Noise Est.	Merging	Total	Time(s)	# Steps
14	LawDB-1	3	100	42	3	1.17	5.45	51.10	57.736	158.78	25
15	LawDB-2	4	10000	90	4	1.81	11.83	127.59	141.24	163.28	45
16	LawDB-3	5	10000	165	5	2.91	23.39	365.13	391.45	188.85	48
17	LawDB-4	9	10000	855	9	20.57	152.09	2436.84	2609.50	271.16	77

Larger Experiments. After seeing that we can automatically synthesize an adaptive range-query attack against LawDB for small domains, we increased the valid range of IDs that are allowed in the database to $[1, 10000]$ and inserted 3, 4, and 8 randomly generated public IDs. In Table 6.3 we see that we are able to synthesize attacks in a reasonable amount of time. We observe that as the number of path constraints increases, the cost of the offline phase grows, with the majority of offline time spent in trace class merging. However, trace class merging is crucial since it reduces the difficulty and the cost of model counting and entropy computations. For instance, for LawDB-4 model counting and entropy computation may be performed over 9 trace classes rather than 855 path constraints, resulting in efficient online attack synthesis.

```
1  public class UDPServerHandler {
2   // Constructors and local variables ...
3   public void channelRead0(int t, int key, int min, int max) {
4     switch (t) {
5     ...
6     case 8: {
7      DSystemHandle sys = new DSystemHandle("127.0.0.1", 6666);
8      List<String> filestoCheck=new ArrayList<String>();
9      final DFileHandle fh1 = new DFileHandle("config.security", sys);
10     filestoCheck.add("config.security");
11     final List<Integer> range =  this.btree.toList(min, max);
12     if (range.size() <= 0 || !this.restricted.isRestricted((int) range.get(0))) {
13       filestoCheck = new ArrayList<String>();
14     }
15     int ind = 0;
16     while (ind < range.size()) {
17       try {
18         final Integer nextkey=(Integer)range.get(ind);
19         if (this.restricted.isRestricted(nextkey)) {
20           BufferedWriter bw = null;
21           FileWriter fw = null;
22           try {
23             String data = "SEARCH_ON_RESTRICTED_KEY_OCCURRED:" +
24                   (Integer.toString((int) nextkey) + "\n");
25             File file = new File(LOGFILE);
26             if (!file.exists()) {
27               file.createNewFile();
28             }
29             fw = new FileWriter(file.getAbsoluteFile(), true);
30             bw = new BufferedWriter(fw);
31             bw.write(data);
32           }
33           catch (IOException e) {
34             e.printStackTrace();
35           }
36           finally {
37             try {
38               if (bw != null)
39                 bw.close();
40               if (fw != null)
41                 fw.close();
42             } catch (IOException ex) {
43               ex.printStackTrace();
44             }
45           }
46           throw new RestrictedAccessException();
47         }
48         if (sys == null) {
49           sys = new DSystemHandle("127.0.0.1", 6666);
50         }
51         at.add("lastaccessinfo.log", Integer.toString(nextkey), nextkey);
52         ++ind;
53       } catch (RestrictedAccessException rae) {
54         for (Integer getkey = (Integer) range.get(ind);
55              this.restricted.isRestricted(getkey) && ind < range.size();
56              getkey = (Integer) range.get(ind)) {
57           if (sys == null) {
58             sys = new DSystemHandle("127.0.0.1", 6666);
59           }
60           if (++ind < range.size()) {
61           }
62         }
63       } finally {
64         if (atx != null) {
65           System.out.println("Cleaning_resources");
66           atx.clean();
67           atx = null;
68         }
69       }
70     }
71     at.clean();
72     sys = new DSystemHandle("127.0.0.1", 6666);
73     break;
74   }
75   // Remaining switch cases...
76 }
```

Figure 6.25: Extracted search function for LawDB.

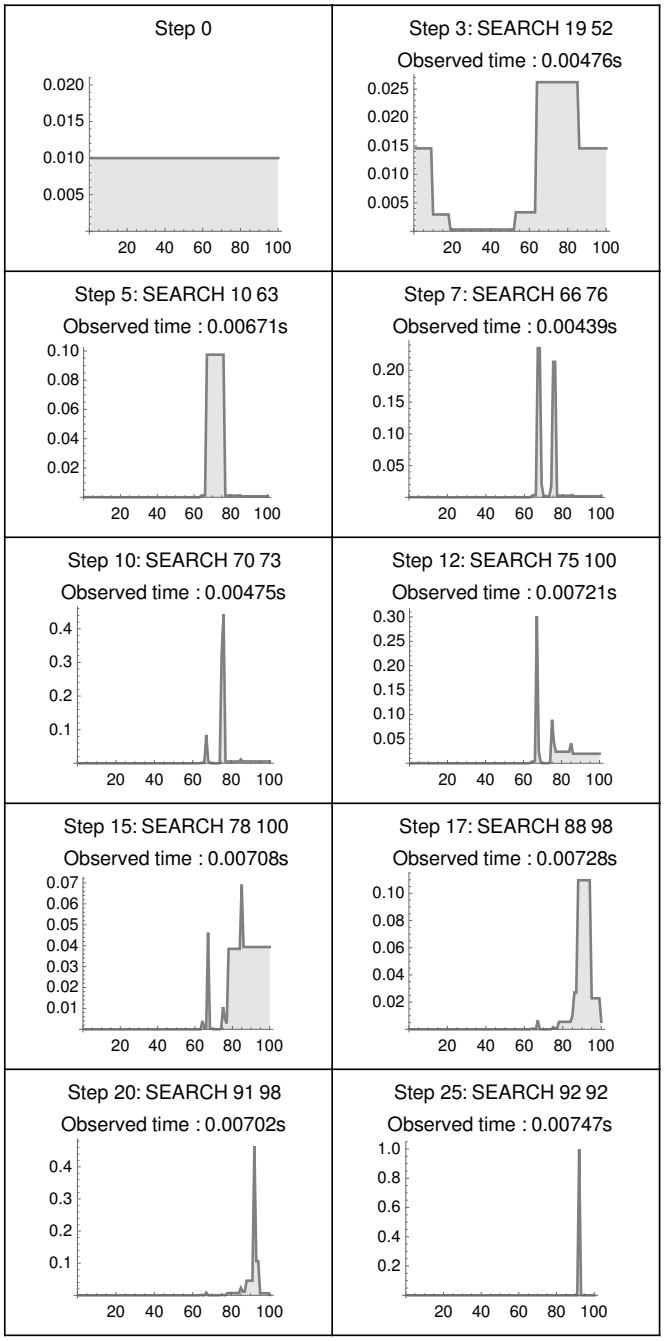


Figure 6.26: Snapshots of $\mathcal{A}$'s belief about the restricted ID for the LawDB–1 case study. In step 5, a noisy observation causes an erroneous belief update, but the synthesized queries eliminate the incorrect belief, to eventually converge to the true value of the restricted ID, 92.

6.7 Chapter Summary

In this chapter I gave details for how to synthesize side-channel attacks. This approach takes into account the probabalistic nature of observations by (1) performing automatic trace class extraction using symbolic execution and dynamic profiling, (2) using Bayesian updates on a dynamically maintained attacker belief distribution $p(h)$, and (3) using *weighted* model counting (where models are weighted by $p(h)$) and numeric maximization ofr choosing attacker inputs at each step. When an attack is synthesized, it provides a proof of exploit for the function under test. I demonstrated that the approach is effective against a set of benchmark programs.

Chapter 7

Related Work

Quantitative information flow (QIF) is a rich field of study incorporating aspects of information theory, program analysis, combinatorics, and game theory. Many results and techniques in the field are based on the pioneering work of Claude Shannon who laid down a quantitative foundation for the study of information [38, 26]. This work borrowed the concept of entropy from thermodynamics and applied it to communication channels, introducing the word 'bit' (binary digit) along the way as a unit of information. QIF leverages information theory to measure the strength of inferencing ability of a malicious adversary in terms of the number of bits that can be learned about secret values. A series of papers by Clark, Hankin, Hunt, and Malacaria laid the groundwork for QIF analysis for software by proposing techniques for measuring the security of imperative-style model languages [102, 103, 104, 105, 106]. These works address the quantification information leakage of high-security program states or variables (intended to be hidden from an attacker) to low-security program states or variables (accessible by an attacker). The core elements of these works were distilled and codified in a later paper which popularized the phrase "quantitative information flow" [107]. Interestingly, around that same time, Kim and Seong independently developed similar entropy-based methods for

measuring the amount of information that flows from safety-critical systems (not directly accessible to operators) to the associated monitoring interfaces (accessible to operators) in the context of engineering system safety and reliability [108, 109, 110]. Important work by Geoffrey Smith provided additional framing, theoretical scaffolding, and useful nomenclature for the study of information flow in software systems [37].

One of the first documented insecure information flows in history that can be called a side channel was kept under wraps by the National Security Agency until 2007 when it was partially declassified. The TEMPEST vulnerability leaked plaintext secret information due to electromagnetic signals generated by the relays in military encryption devices [12]. More recently, it has been shown that dot-matrix printer sounds can be correlated with the content of the resulting printed page [13], inter-keypress timing differences can be measured by a network eavesdropper to determine an SSH user's encrypted password [14], and fairly simple radio equipment can be combined with sophisticated processing on the distortion of wifi signals from a person typing on a laptop to reveal their keystrokes from a distance with high accuracy [15]. However, unlike the software side channels addressed in this dissertation, these side-channels are more related to the electro-mechanical properties of the physical devices and the interaction of the human user with the system.

Turning our attention closer to programmatic side channels, Paul Kocher showed that one can extract secret keys from a cryptographic device pair (e.g. a smart card and reader), since different instructions executed by the microprocesser have different power usage profiles. Measuring these profiles with standard signal processing equipment can reveal cryptographic keys used during DES, AES, and RSA encryption [16], Later, Brumley showed that RSA keys can be adaptively leaked by measuring processing times through a network [7].

The side-channels just described were discovered through manual effort, analysis, and experimentation. Automatic discovery of side-channel vulnerabilities remains a challeng-

ing and active area of research. Promising advances in this area have been made in which the framework provided by QIF has been adapted to the field of side-channel analysis by introducing the notion of an *observable* in addition to the *high*-security and *low*-security variables. There is a considerable amount of fairly recent work related to quantifying side-channel information leakage for a single run of a program [111, 112, 113, 114, 42, 115, 116], but none of these considers multiple runs of the program. Work in [33] addresses computing side channel leakage for multiple runs of a program using symbolic execution but does not address adaptively chosen input sequences. That work, and many of the aforementioned works make use of symbolic execution for automatically analyzing the source code of the program [21, 30].

A model for adaptive attacks was introduced in [93], which assumes noiseless observations and generates attack strategies for any possible secret by enumerating all possible program–adversary behaviors and all partitions of the secret induced by the public input. A 2018 CSF paper on side-channel analysis gives a static technique for measuring information leakage for a single run, in the presence of programmatic randomness, using symbolic sampling methods that provide upper and lower bounds on information flow. This work also makes use of Barvinok for model counting [117].

An information flow model based on Bayesian belief updates for an adversary who makes observations of a probabilistic program was presented in [118]. Their work specifically addresses single runs of a program, but the authors indicate that Bayesian belief updates may be used over multiple runs of the program. In [119], the authors illustrate methods for detecting the possibility of side channels in client-server application traffic. Work in [120] also addresses DARPA STAC programs. Their effort is concentrated on showing safety properties of non-vulnerable programs and is able to indicate possible side channel vulnerabilities by detecting observationally imbalanced program branches, but does not generate attacks. Two works [121, 122] describe how to quantify infor-

mation leakage in interactive systems in which the secret changes over repeated runs of the application. In [97] a method is given to quantify the trade-off between program reliability and security when adding noise to a program in order to reduce side channel vulnerability. There is also work on program transformations that remove side-channel vulnerabilities [123]. In [124] the problem of performing Bayesian inference via weighted model counting is described with applications to statistical prediction. In [125] a method is given for performing precise quantitative information flow analysis using symbolic weighted model counting.

Model counting is a crucial component of quantitative program analysis, including side-channel detection and measurement. This dissertation addressed information leakage quantification for string manipulating programs in Chapter 4 using the automata-based counting methods of Chapter 3. There has been significant amount of work on string constraint solving in recent years [126, 51, 127, 128, 129, 52, 53, 54, 130, 55]; however none of these address the model-counting problem. Due to the importance of model counting in quantitative program analyses, model counting constraint solvers are gaining increasing attention. S3# and SMC are the only other model-counting string constraint solvers that we are aware of [131, 44]. Our approach to model counting is strictly more precise than SMC. SMC cannot propagate string values across logical connectives which reduces its precision during model counting, whereas we can handle logical connectives without losing precision.

ABC builds on the automata-based string analysis tool Stranger [70, 71, 72] determined to be the best string solver in terms of precision and efficiency in a recent empirical study [132]. While linear algebraic methods for counting paths in a graph are well established [61], my approach was the first to implement those methods for the purpose of parameterized model counting for string and integer constraints. There has been earlier work on integer constraint model counting by counting paths in numeric DFA [50],

but this earlier approach can only count models when there are finitely many models. LattE [56] is a model counting constraint solver for linear integer arithmetic that has been used in several quantitative program analyses [40, 133, 32]. LattE uses the polynomial-time Barvinok algorithm [43] for integer lattice point enumeration. In addition, there is a separate model counting library named Barvinok, which implements Barvinok's algorithm [57] that has also been used for QIF analysis [117, 125]. Finally, there are randomized sampling based methods for approximating the number of solutions to a constraint. SMTApproxMC [134] is a model counting constraint solver for the theory of fixed-width words, and it uses a different generic approach for model-counting which is based on hash-function sampling [135]. For a broad survey of methods for Boolean model counting, we refer the reader to Chapter 20 of the Handbook of Satisfiability [60].

Chapter 8

Conclusion

Side-channel vulnerabilities are notoriously insidious, creeping into software written by well-intentioned developers. Why do these types of security flaws arise? In the (possibly apocryphal) words of Donald Knuth, "Premature optimization is the root of all evil." While this is a rather extreme sentiment, there is a thread of truth in it when it comes to side-channels. For instance, in the case of segmented oracles a side-channel vulnerability arises due to software developers who are trying to do what they were taught—always make your code as efficient as possible. So, when writing code that compares two arrays, for instance, they will return from the function as soon as it is possible to determine the correct return value. However, this increase in efficiency results in an unintended security vulnerability. We can see that, at least in segment oracle side channels, there is a trade-off between the efficiency of the code and the security with respect to side channel observations.

The widespread existence of the the segment oracle pattern—it is in C's `memcmp`, OAuth, Java's `Array.Equals()`, Python's string equality comparison—is one piece of evidence to support the idea that side channel vulnerabilities are not at all obvious to developers who are writing the code. Thus, in this concluding chapter, I would like to

emphasize the importance of (1) developing theories and tools for helping experts and non-experts detect side-channel vulnerabilities in their code and (2) encouraging students who are learning to program to think critically and carefully about the trade-offs made when optimizing code that might operate on sensitive data.

Overall, this dissertation has drawn upon existing works in quantitative information flow, side-channel analysis, model counting, and automatic software verification and I pushed the state-of-the-art in software side-channel analysis by creating and demonstrating new methods for quantifying, discovering, and synthesizing side-channel attacks. Specifically I introduced (1) automata-based model counting techniques, (2) novel QIF analysis for segmented oracle side channels, (3) offline static side-channel attack synthesis and quantification, and (4) online attack synthesis that accounts for noisy and networked systems. My attack synthesis approaches made use of symbolic polytope-based model counting, information-theoretic objective functions, and numeric optimization. I demonstrated the effectiveness of these 4 approaches for quantitative program analyses through experimentation.

Bibliography

[1] J. Ziv and A. Lempel, *A universal algorithm for sequential data compression*, *IEEE Transactions on Information Theory* **23** (May, 1977) 337–343.

[2] A. Research, *DARPA STAC project public software repository*, 2017.

[3] M. Joye, *Basics of Side-Channel Analysis*, in *Cryptographic Engineering*, ch. 13, pp. 367–382. 2009.

[4] S. Chen, R. Wang, X. Wang, and K. Zhang, *Side-channel leaks in web applications: A reality today, a challenge tomorrow*, in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 191–206, IEEE Computer Society, 2010.

[5] "Xbox 360 timing attack." `http://beta.ivc.no/wiki/index.php/Xbox_360_Timing_Attack`, 2007.

[6] N. Lawson, "Timing attack in google keyczar library." `https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/`, 2009.

[7] D. Brumley and D. Boneh, *Remote Timing Attacks Are Practical*, in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2003.

[8] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, *Spectre attacks: Exploiting speculative execution*, *CoRR* **abs/1801.01203** (2018) [arXiv:1801.0120].

[9] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, *Meltdown*, *CoRR* **abs/1801.01207** (2018) [arXiv:1801.0120].

[10] P. Gray, *And bomb the anchovies*, *Time Magazine* **136** (aug, 1990).

[11] N. Zaidenberg and A. Resh, *Timing and Side Channel Attacks*, pp. 183–194. Springer International Publishing, Cham, 2015.

[12] N. S. Agency, "Tempest: A signal problem." https://www.nsa.gov/news-features/declassified-documents/cryptologic-spectrum/assets/files/tempest.pdf, 1972.

[13] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, *Acoustic side-channel attacks on printers*, in *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, (Berkeley, CA, USA), pp. 20–20, USENIX Association, 2010.

[14] D. X. Song, D. A. Wagner, and X. Tian, *Timing analysis of keystrokes and timing attacks on SSH*, in *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*, 2001.

[15] K. Ali, A. X. Liu, W. Wang, and M. Shahzad, *Keystroke recognition using wifi signals*, in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, (New York, NY, USA), pp. 90–102, ACM, 2015.

[16] P. C. Kocher, J. Jaffe, and B. Jun, *Differential power analysis*, in *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pp. 388–397, 1999.

[17] C. Hale, "A lesson in timing attacks (or, dont use messagedigest.isequals)." https://codahale.com/a-lesson-in-timing-attacks/, 2009.

[18] J. S. Daniel Mayer, "Time trial: Racing towards practical remote timing attacks." https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/TimeTrial.pdf, 2014.

[19] J. Kelsey, *Compression and information leakage of plaintext*, in *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*, pp. 263–276, 2002.

[20] J. Rizzo and T. Duong, *The crime attack*, Ekoparty Security Conference, 2012.

[21] J. C. King, *Symbolic execution and program testing*, *Commun. ACM* **19** (July, 1976) 385–394.

[22] Q.-S. Phan, *Symbolic Execution as DPLL Modulo Theories*, in *2014 Imperial College Computing Student Workshop*, vol. 43 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 58–65, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

[23] C. Cadar and K. Sen, *Symbolic Execution for Software Testing: Three Decades Later*, *Commun. ACM* **56** (Feb., 2013) 82–90.

[24] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, *Effective lattice point counting in rational convex polytopes*, *Journal of Symbolic Computation* **38** (2004), no. 4 1273 – 1302. Symbolic Computation in Algebra and Geometry.

[25] A. Aydin, L. Bang, and T. Bultan, *Automata-based model counting for string constraints*, in *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, pp. 255–272, 2015.

[26] T. M. Cover and J. A. Thomas, *Elements of information theory*. Wiley-Interscience, New York, NY, USA, 1991.

[27] P. Malacaria, *Quantitative information flow: from theory to practice?*, in *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, (Berlin, Heidelberg), pp. 20–22, Springer-Verlag, 2010.

[28] N. Lawson, "Optimized memcmp leaks useful timing differences." `https://rdist.root.org/2010/08/05/optimized-memcmp-leaks-useful-timing-differences/`, 2010.

[29] L. De Moura and N. Bjørner, *Z3: an efficient SMT solver*, in *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[30] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, *Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis*, *Automated Software Engineering* (2013) 1–35.

[31] K. S. Luckow, C. S. Pasareanu, M. B. Dwyer, A. Filieri, and W. Visser, *Exact and approximate probabilistic symbolic execution for nondeterministic programs*, in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pp. 575–586, 2014.

[32] A. Filieri, C. S. Pasareanu, and W. Visser, *Reliability analysis in symbolic pathfinder*, in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pp. 622–631, 2013.

[33] C. S. Păsăreanu, Q.-S. Phan, and P. Malacaria, *Multi-run side-channel analysis using Symbolic Execution and Max-SMT*, in *Proceedings of the 2016 IEEE 29th Computer Security Foundations Symposium*, CSF '16, (Washington, DC, USA), IEEE Computer Society, 2016.

[34] D. J. D. Hughes and V. Shmatikov, *Information hiding, anonymity and privacy: a modular approach*, *Journal of Computer Security* **12** (2004), no. 1 3–36.

[35] M. Backes and B. Pfitzmann, *Computational probabilistic noninterference*, *Int. J. Inf. Sec.* **3** (2004), no. 1 42–60.

[36] R. Ensafi, J. C. Park, D. Kapur, and J. R. Crandall, *Idle port scanning and non-interference analysis of network protocol stacks using model checking*, in *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pp. 257–272, 2010.

[37] G. Smith, *On the foundations of quantitative information flow*, in *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, pp. 288–302, 2009.

[38] C. Shannon, *A mathematical theory of communication*, *Bell System Technical Journal* **27** (July, October, 1948) 379–423, 623–656.

[39] T. M. Cover and J. A. Thomas, *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006.

[40] Q.-S. Phan, *Model Counting Modulo Theories*. PhD thesis, Queen Mary University of London, 2015.

[41] J. R. J. Bayardo and J. D. Pehoushek, *Counting Models Using Connected Components*, in *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pp. 157–162, AAAI Press, 2000.

[42] Q.-S. Phan and P. Malacaria, *Abstract Model Counting: A Novel Approach for Quantification of Information Leaks*, in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, (New York, NY, USA), pp. 283–292, ACM, 2014.

[43] A. I. Barvinok, *A Polynomial Time Algorithm for Counting Integral Points in Polyhedra When the Dimension is Fixed*, *Math. Oper. Res.* **19** (1994), no. 4 769–779.

[44] L. Luu, S. Shinde, P. Saxena, and B. Demsky, *A model counter for constraints over unbounded strings*, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, p. 57, 2014.

[45] V. Klebanov, *Precise Quantitative Information Flow Analysis Using Symbolic Model Counting*, in *Proceedings, International Workshop on Quantitative Aspects in Security Assurance (QASA)* (F. Martinelli and F. Nielson, eds.), 2012.

[46] K. von Gleissenthall, B. Köpf, and A. Rybalchenko, *Symbolic polytopes for quantitative interpolation and verification*, in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), (Cham), pp. 178–194, Springer International Publishing, 2015.

[47] M. Chavira and A. Darwiche, *On probabilistic inference by weighted model counting*, *Artificial Intelligence* **172** (2008), no. 6 772 – 799.

[48] T. Sang, P. Beame, and H. A. Kautz, *Performing bayesian inference by weighted model counting*, in *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pp. 475–482, 2005.

[49] W. Pugh, *Counting solutions to presburger formulas: How and why*, in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, (New York, NY, USA), pp. 121–134, ACM, 1994.

[50] E. Parker and S. Chatterjee, *An automata-theoretic algorithm for counting solutions to presburger formulas*, in *Compiler Construction* (E. Duesterwald, ed.), (Berlin, Heidelberg), pp. 104–119, Springer Berlin Heidelberg, 2004.

[51] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, *Hampi: a solver for string constraints*, in *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, pp. 105–116, 2009.

[52] Y. Zheng, X. Zhang, and V. Ganesh, *Z3-str: A z3-based string solver for web application analysis*, in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pp. 114–124, 2013.

[53] G. Li and I. Ghosh, *PASS: string solving with parameterized array and interval automaton*, in *Proceedings of the 9th International Haifa Verification Conference (HVC)*, pp. 15–31, 2013.

[54] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, *String constraints for verification*, in *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, pp. 150–166, 2014.

[55] M. Trinh, D. Chu, and J. Jaffar, *S3: A symbolic string solver for vulnerability detection in web applications*, in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1232–1243, 2014.

[56] "LattE." `http://www.math.ucdavis.edu/~latte/`.

[57] "Barvinok library." `http://garage.kotnet.org/~skimo/barvinok/`.

[58] S. Verdoolaege, *The barvinok model counter*, 2017.

[59] E. Birnbaum and E. L. Lozinskii, *The good old Davis-Putnam procedure helps counting models*, *J. Artif. Int. Res.* **10** (June, 1999) 457–477.

[60] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[61] R. P. Stanley, *Enumerative Combinatorics: Volume 1*. Cambridge University Press, New York, NY, USA, 2nd ed., 2011.

[62] P. Flajolet and R. Sedgewick, *Analytic Combinatorics*. Cambridge University Press, New York, NY, USA, 1 ed., 2009.

[63] N. Chomsky and M. P. Schtzenberger, *The algebraic theory of context-free languages*, 01, 1970.

[64] A. I. Barvinok, *A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed*, *Math. Oper. Res.* **19** (Nov., 1994) 769–779.

[65] L. Bang, A. Aydin, Q.-S. Phan, C. S. Pasareanu, and T. Bultan, *String analysis for side channels with segmented oracles*, in *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2016.

[66] N. Biggs, *Algebraic Graph Theory*. Cambridge Mathematical Library. Cambridge University Press, 1993.

[67] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[68] J. L. Gross, J. Yellen, and P. Zhang, *Handbook of Graph Theory, Second Edition*. Chapman & Hall/CRC, 2nd ed., 2013.

[69] D. E. Knuth, *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[70] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra, *Symbolic string verification: An automata-based approach*, in *Proceedings of the 15th International SPIN Workshop on Model Checking Software (SPIN)*, pp. 306–324, 2008.

[71] F. Yu, M. Alkhalaf, and T. Bultan, *Stranger: An automata-based string analysis tool for php*, in *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 154–157, 2010.

[72] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra, *Automata-based symbolic string analysis for vulnerability detection*, *Formal Methods in System Design* **44** (2014), no. 1 44–70.

[73] BRICS, "The MONA project." `http://www.brics.dk/mona/`.

[74] I. Wolfram Research, *Mathematica*, 2014.

[75] C. Bartzis and T. Bultan, *Efficient symbolic representations for arithmetic constraints in verification*, *Int. J. Found. Comput. Sci.* **14** (2003), no. 4 605–624.

[76] F. Weimer, "Defeating memory comparison timing oracles." `https://access.redhat.com/blogs/766093/posts/878863/`, 2014.

[77] N. Lawson, *Side-channel attacks on cryptographic software*, *IEEE Security and Privacy* **7** (Nov., 2009) 65–68.

[78] "A few important facts regarding oauth security." `http://oauthlib.readthedocs.io/en/latest/oauth1/security.html`, 2012.

[79] "Oauth protocol hmac byte value calculation timing disclosure weakness." `https://osvdb.info/OSVDB-97562`, 2013.

[80] T. Nelson, "Widespread timing vulnerabilities in openid implementations." `http://lists.openid.net/pipermail/openid-security/2010-July/001156.html`, 2010.

[81] A. Filieri, C. S. Păsăreanu, and W. Visser, *Reliability analysis in symbolic pathfinder*, in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 622–631, IEEE Press, 2013.

[82] B. Köpf and D. Basin, *An Information-theoretic Model for Adaptive Side-channel Attacks*, in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, (New York, NY, USA), pp. 286–296, ACM, 2007.

[83] P. Malacaria and H. Chen, *Lagrange multipliers and maximum information leakage in different observational models*, in *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, (New York, NY, USA), pp. 135–146, ACM, 2008.

[84] G. Smith, *On the Foundations of Quantitative Information Flow*, in *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS '09, (Berlin, Heidelberg), pp. 288–302, Springer-Verlag, 2009.

[85] Q. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, *Synthesis of adaptive side-channel attacks*, *IACR Cryptology ePrint Archive* **2017** (2017) 401.

[86] R. Nieuwenhuis and A. Oliveras, *On SAT Modulo Theories and Optimization Problems*, in *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, (Berlin, Heidelberg), pp. 156–169, Springer-Verlag, 2006.

[87] M. H. Liffiton and A. Malik, *Enumerating Infeasibility: Finding Multiple MUSes Quickly*, pp. 160–175. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[88] W. Research, *Mathematica 11.0*, 2016.

[89] S. Das and P. N. Suganthan, *Differential evolution: A survey of the state-of-the-art*, *IEEE Trans. Evolutionary Computation* **15** (2011), no. 1 4–31.

[90] "DARPA STAC program." `http://www.darpa.mil/program/space-time-analysis-for-cybersecurity`.

[91] "Netty library." `http://netty.io/`.

[92] A. Sabelfeld and A. C. Myers, *Language-based information-flow security*, *IEEE Journal on Selected Areas in Communications* **21** (2003), no. 1 5–19.

[93] B. Köpf and D. A. Basin, *An information-theoretic model for adaptive side-channel attacks*, in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007* (P. Ning, S. D. C. di Vimercati, and P. F. Syverson, eds.), pp. 286–296, ACM, 2007.

[94] Wolfram Research, *Wolfram Language Documentation: NMaximize*, 2017.

[95] M. Rosenblatt, *Remarks on some nonparametric estimates of a density function*, *Ann. Math. Stat.* **27** (1956), no. 3 832–837.

[96] E. Parzen, *On estimation of a probability density function and mode*, *Ann. Math. Statist.* **33** (1962), no. 3 1065–1076.

[97] T. M. Ngo and M. Huisman, *Quantitative Security Analysis for Programs with Low Input and Noisy Output*, pp. 77–94. Springer International Publishing, Cham, 2014.

[98] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, *Thwarting cache side-channel attacks through dynamic software diversity*, in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, The Internet Society, 2015.

[99] DARPA, *The space-time analysis for cybersecurity (STAC) project*, 2015.

[100] Wolfram Research, *Mathematica Version 11*, 2017. Champaign, IL, 2017.

[101] Wolfram Research, *Wolfram MathWorld: DifferentialEvolution*, 2017.

[102] D. Clark, C. Hankin, and S. Hunt, *Information flow for algol-like languages*, *Comput. Lang. Syst. Struct.* **28** (Apr., 2002) 3–28.

[103] D. Clark, S. Hunt, and P. Malacaria, *Quantified Interference for a While Language*, *Electron. Notes Theor. Comput. Sci.* **112** (Jan., 2005) 149–166.

[104] D. Clark, S. Hunt, and P. Malacaria, *Quantitative Analysis of the Leakage of Confidential Data* , *Electronic Notes in Theoretical Computer Science* **59** (2002), no. 3 238 – 251. QAPL'01, Quantitative Aspects of Programming Laguages (Satellite Event of {PLI} 2001).

[105] D. Clark, S. Hunt, and P. Malacaria, *Quantified interference: information theory and information flow*, in *Presented at Workshop on Issues in the Theory of Security (WITS04*, p. 04, 2004.

[106] P. Malacaria, *Assessing security threats of looping constructs*, in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, (New York, NY, USA), pp. 225–235, ACM, 2007.

[107] D. Clark, S. Hunt, and P. Malacaria, *Quantitative Information Flow, Relations and Polymorphic Types*, *J. Log. and Comput.* **15** (Apr., 2005) 181–199.

[108] M. C. Kim and P. H. Seong, *A computational model for knowledge-driven monitoring of nuclear power plant operators based on information theory*, *Rel. Eng. & Sys. Safety* **91** (2006), no. 3 283–291.

[109] J. H. Kim and P. H. Seong, *A quantitative approach to modeling the information flow of diagnosis tasks in nuclear power plants*, *Rel. Eng. & Sys. Safety* **80** (2003), no. 1 81–94.

[110] H. G. Kang and P. Seong, *Information theoretic approach to man-machine interface complexity evaluation*, *IEEE Trans. Systems, Man, and Cybernetics, Part A* **31** (2001), no. 3 163–171.

[111] M. Backes, B. Kopf, and A. Rybalchenko, *Automatic Discovery and Quantification of Information Leaks*, in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, (Washington, DC, USA), pp. 141–153, IEEE Computer Society, 2009.

[112] J. Heusser and P. Malacaria, *Quantifying information leaks in software*, in *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, (New York, NY, USA), pp. 261–269, ACM, 2010.

[113] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu, *Symbolic Quantitative Information Flow*, *SIGSOFT Softw. Eng. Notes* **37** (Nov., 2012) 1–5.

[114] V. Klebanov, N. Manthey, and C. Muise, *SAT-Based Analysis and Quantification of Information Flow in Programs*, in *Quantitative Evaluation of Systems*, vol. 8054 of *Lecture Notes in Computer Science*, pp. 177–192. Springer Berlin Heidelberg, 2013.

[115] Q.-S. Phan, P. Malacaria, C. S. Păsăreanu, and M. d'Amorim, *Quantifying Information Leaks Using Reliability Analysis*, in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, (New York, NY, USA), pp. 105–108, ACM, 2014.

[116] Q.-S. Phan and P. Malacaria, *All-Solution Satisfiability Modulo Theories: applications, algorithms and benchmarks*, in *Proceedings of the 2015 Tenth International Conference on Availability, Reliability and Security*, ARES '15, (Washington, DC, USA), IEEE Computer Society, 2015.

[117] P. Malacaria, M. H. R. Khouzani, C. S. Pasareanu, Q. Phan, and K. S. Luckow, *Symbolic side-channel analysis for probabilistic programs*, *IACR Cryptology ePrint Archive* **2018** (2018) 329.

[118] M. R. Clarkson, A. C. Myers, and F. B. Schneider, *Belief in information flow*, in *Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, CSFW '05, (Washington, DC, USA), pp. 31–45, IEEE Computer Society, 2005.

[119] P. Chapman and D. Evans, *Automated black-box detection of side-channel vulnerabilities in web applications*, in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), pp. 263–274, ACM, 2011.

[120] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, *Decomposition instead of self-composition for k-safety*, Nov., 2016.

[121] P. Mardziel, M. S. Alvim, M. W. Hicks, and M. R. Clarkson, *Quantifying information flow for dynamic secrets*, in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pp. 540–555, 2014.

[122] M. S. Alvim, M. E. Andrés, and C. Palamidessi, *Information Flow in Interactive Systems*, pp. 102–116. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[123] H. Mantel and A. Starostin, *Transforming Out Timing Leaks, More or Less*, pp. 447–467. Springer International Publishing, Cham, 2015.

[124] M. Chavira and A. Darwiche, *On probabilistic inference by weighted model counting*, *Artif. Intell.* **172** (Apr., 2008) 772–799.

[125] V. Klebanov, *Precise quantitative information flow analysis - a symbolic approach*, *Theor. Comput. Sci.* **538** (2014) 124–139.

[126] P. Hooimeijer and W. Weimer, *A decision procedure for subset constraints over regular languages*, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 188–198, 2009.

[127] P. Hooimeijer and W. Weimer, *Solving string constraints lazily*, in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 377–386, 2010.

[128] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, *A symbolic execution framework for javascript*, in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[129] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. C. Rinard, *Word equations with length constraints: What's decidable?*, in *Proceedings of the 8th International Haifa Verification Conference (HVC)*, pp. 209–226, 2012.

[130] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, *A DPLL(T) theory solver for a theory of strings and regular expressions*, in *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, pp. 646–662, 2014.

[131] M.-T. Trinh, D.-H. Chu, and J. Jaffar, *Model counting for recursively-defined strings*, in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, Proceedings, Part II*, pp. 399–418, 2017.

[132] S. Kausler and E. Sherman, *Evaluation of string constraint solvers in the context of symbolic execution*, in *Proceedings of the 29th ACM/IEEE International Conference on Automated software engineering (ASE)*, pp. 259–270, 2014.

[133] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu, *Symbolic quantitative information flow*, *SIGSOFT Softw. Eng. Notes* **37** (2012), no. 6 1–5.

[134] S. Chakraborty, K. S. Meel, R. Mistry, and M. Y. Vardi, *Approximate probabilistic inference via word-level counting*, in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pp. 3218–3224, 2016.

[135] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, *Distribution-aware sampling and weighted model counting for SAT*, in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pp. 1722–1730, 2014.